Mrežna usluga za rješavanje problema redova čekanja

Perković, Tedi

Sveučilište Jurja Dobrile u Puli

Odjel za informacijsko-komunikacijske tehnologije

TEDI PERKOVIĆ

MREŽNA USLUGA ZA RJEŠAVANJE PROBLEMA REDOVA ČEKANJA

Diplomski rad

Pula, lipanj 2018. godine

Sveučilište Jurja Dobrile u Puli

Odjel za informacijsko-komunikacijske tehnologije

TEDI PERKOVIĆ

MREŽNA USLUGA ZA RJEŠAVANJE PROBLEMA REDOVA ČEKANJA

Diplomski rad

JMBAG: 0016057612, redoviti student

Studijski smjer: Informatika

Predmet: Izrada informatičkih projekata

Znanstveno područje: Društvene znanosti

Znanstveno polje: Informacijske i komunikacijske znanosti

Znanstvena grana: Informacijski sustavi i informatologija

Mentor: doc. dr. sc. Siniša Sovilj

Pula, 10. lipanj 2018. godine

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisan Tedi Perković, ovime izjavljujem da je ovaj diplomski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio diplomskog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, 10.06.2018. godine

IZJAVA

o korištenju autorskog djela

Ja, Tedi Perković, dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom „Mrežna usluga za upravljanje redovima čekanja“ koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, 10. lipnja 2018. godine

Potpis

_____

SVEUČILIŠTE JURJA DOBRILE U PULI
ODJEL ZA INFORMACIJSKO-KOMUNIKACIJSKE TEHNOLOGIJE

Pula, 1. ožujka 2017.

DIPLOMSKI ZADATAK

Pristupnik:	**Tedi Perković (0016057412)**
Studij:	Sveučilišni diplomski studij Informatike
Naslov (hrv.):	**Mrežna usluga za rješavanje problema redova čekanja**
Naslov (eng.):	Queue managing web service
Opis zadatka:	Zadatak je istražiti i razviti REST uslugu u Java Spring programskom okviru, odnosno API za pristup i upravljanje podacima, koji će se koristiti na kritičnim mjestima gdje je potrebno smanjivanje ili uklanjanje redova čekanja. Uslugu prilagoditi za primijenu u raznim ustanovama kao što su sveučilišta, policija, ambulante i slično. Omogućiti korisnicima registraciju, pristup sustavu putem postojećeg Google ili Facebook korisničkog računa, listu ustanova koje korisnik može izabrati, pregled i uzimanje broja s određenoga reda u ustanovi te provjeru predviđenog vremena čekanja za pojedini red u ustanovi.

Zadatak uručen pristupniku: 1. ožujka 2017.
Rok za predaju rada: 1. veljače 2018.

Mentor:
Siniša Sovilj
_____
doc.dr.sc. Siniša Sovilj

SADRŽAJ

UVOD

Čekanje u redu, bilo na šalteru u banci ili za upis na godinu studija, stvara stres i oduzima slobodno vrijeme. Štoviše to neke čini bijesnima, neke arogantnima, a najčešće je frustrirajuće za sve korisnike usluga koji se našli u dugačkome redu. U svakom slučaju čekanje nije pozitivno iskustvo budući da troši slobodno vrijeme, a za današnji brzi način života vrijeme je dragocjeno. Često korisnici ne žele čekati u redu satima i odgađaju obavljanje jednostavnih administracijskih postupaka te su zbog toga frustrirani i nervozni.

Otuda se pojavila ideja o sustavu koji bi problem loše iskorištenog vremena uvelike smanjio ili, u nekim slučajevima, potpuno uklonio. Internet i mobilni uređaji danas su svakodnevnica te se ideja za sustav za upravljanje redovima čekanja može jednostavno realizirati preko tih medija. Kako se ne bi ograničilo samo na određeni mobilni uređaj, aplikaciju ili web stranicu ideja je napraviti web servis koji će biti dislociran te dostupan na Internetu preko kojega će bilo koji mobilni uređaji, aplikacije ili web stranice moći pristupiti te izvršavati akcije vezane za uzimanje rednoga broja. Ovisno o ustanovi i redu moguće je vidjeti trenutni redni broj koji ovisi o mogućnostima rješavanja posla šalterskih službenika.

Rad se sastoji od pet poglavlja isključujući uvod, zaključak i sažetak. Prvo poglavlje uvodi u problem redova čekanja i razloge zašto bi se taj problem trebao pokušati umanjiti. Opisana je ideja o web servisu za upravljanje redovima čekanja, njegove funkcionalnosti te use case dijagram.

Drugo poglavlje definira što su web servisi općenito te detaljnije što znači REST sa principima zašto se treba pratiti njegova konvencija.

Istraživanje potrebnih tehnologija za izradu web servisa je fokus trećeg poglavlja. Sastoji se od četiri potpoglavlja koja definiraju razloge za odabir određenih tehnologija za izradu web servisa sa detaljnijom razradom specifične tehnologije.

Planiranje i projektiranje pozadinskog servisa sa objašnjenjima koje su tehnologije potrebne za izgradnju funkcionalnog sustava će sadržavati četvrto poglavlje.

U petom i posljednjem poglavlju je primjena odabranih tehnologija. Poglavlje se sastoji od osam potpoglavlja. Kroz prvo potpoglavlje prikazani su potrebni koraci za uspješnu konfiguraciju projekta sa korisnim dodacima koji pomažu u izradi projekta. Prednosti dislociranosti web servisa i baze podataka razrađeni su u drugom potpoglavlju sa potrebnim tehnologijama za ostvarenje tog cilja. Opis i izgled entiteta podataka nalazi se u trećem potpoglavlju. Četvrto potpoglavlje sadrži detaljan opis implementacije OAuth2 standarda na web servisu upravljanja redovima čekanja. Načini registracije korisnika i njegova prijava u sustav opisani su u petom potpoglavlju. Krajnje točke sustava sa njihovim akcijama za korisnika, ustanove i redove detaljno su opisani u potpoglavljima šest, sedam i osam.

1 UPRAVLJANJE REDOVIMA ČEKANJA

Redovi i gužve koje se stvaraju u javnim i privatnim ustanovama rezultiraju nezadovoljstvom korisnika i djelatnika. Nitko ne voli čekati te bi se u današnje užurbano vrijeme svakodnevne posjete ustanovama trebale odvijati puno brže. Ideja je napraviti web servis s kojom bi se redovi smanjili ili potpuno eliminirali.

Web servis koji na jednostavan način omogućuje uzimanje virtualnog broja u redu, praćenje stanja u redu i pravovremene obavijesti o tome kada smo na redu – sve u svrhu bržeg i lakšeg obavljanja svakodnevnih zadataka poput plaćanja računa ili posjeta liječniku. Jednostavno korištenje, bez potrebe instalacije od strane korisnika te mogućnost korištenja bez obzira na operativni sustav uređaja preko kojega se pristupa servisu rješenje je za sve ustanove u kojima se stvaraju svima omraženi redovi čekanja.

Ljudi više neće morati stajati u redu niti čekati u poslovnici, točnu informaciju o vremenu dolaska na red korisnici će moći provjeriti putem svojih mobilnih telefona. Ovaj inovativan pristup upravljanja redovima čekanja uvelike će doprinijeti boljem iskorištavanju slobodnog vremena korisnika s jedne strane, dok će s druge strane voditelji poslovnica moći pratiti aktivnost u poslovnicama te prema tome organizirati i optimizirati rad šaltera.

Omogućava zaposlenicima da se bolje pripreme za upite i zahtjeve korisnika (što rezultira manjim obujmom posla) u čemu se očituje funkcionalnost ovakvog softverskog rješenja.

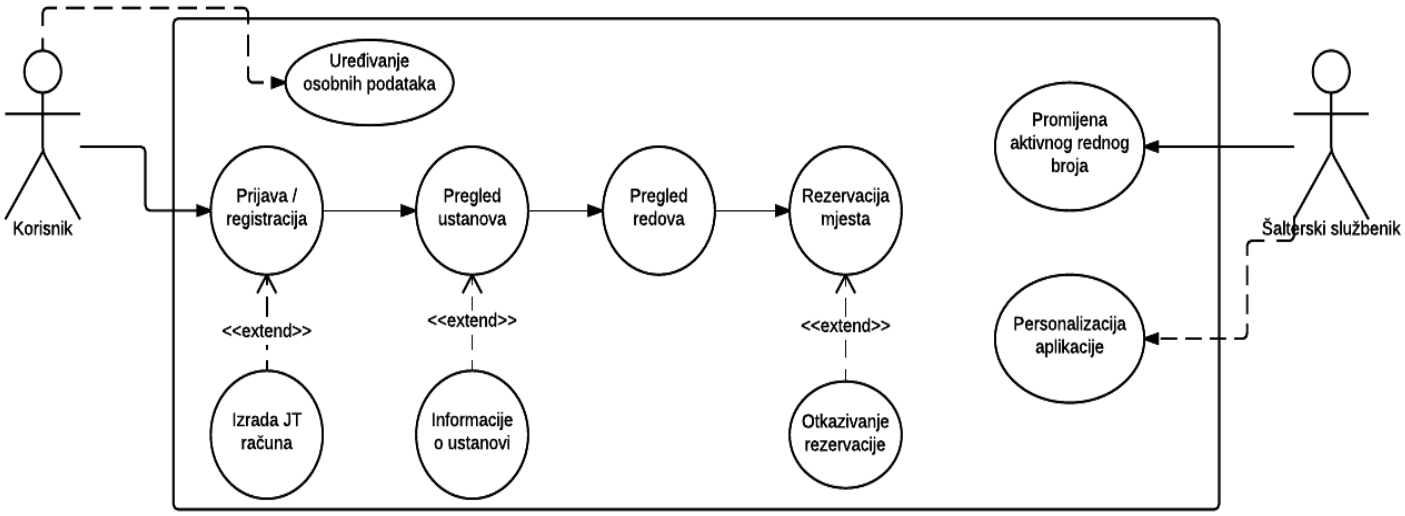
Ciljano tržište su sve ustanove koje imaju potrebu ukloniti ili smanjiti redove čekanja te svaka osoba koja ne voli čekati u redu. Konkretnije, javne i privatne ustanove te korisnici koji spadaju u populaciju od 18 do 65 godina. Ustanove koje bi mogle imati koristi od web servisa su: fakulteti, banke, pošte, bolnice, domovi zdravlja, poslovnice telekoma, državne ustanove (MUP) i slično.

Funkcionalnosti web servisa

- **Prijava** – Pri uzimanju broja korisniku će biti ponuđena tri načina za login: putem Facebook profila, putem Google+ profila ili putem profila. Odabirom jednog od prva dva slučaja korisnik će se ulogirati sa već postojećim profilom pripadajuće društvene mreže, dok će treći način nuditi registraciju putem servisa. Registracija za profil je vrlo jednostavna. Potrebno je unijeti svoje ime i prezime te željeni email i lozinku.
- **Ustanove** – prikaz svih ustanova koje su podržane u web servisu. Moguće je vidjeti informacije o ustanovi kao što su ime, broj telefona, adresa.
- **Red** – prikaz redova u odabranoj ustanovi. Kod redova mogu se vidjeti informacije o tome koliko je korisnika u redu.

Na slici ispod objašnjen je proces rada sustava za upravljanje redovima čekanja.

- Korisnik želi uzeti broj. Prvo se prijavi ili registrira (izrađujući račun). Nakon prijave pregleda ustanove te odabere jednu, zatim pregleda redove te si rezervira mjesto u tom redu. Ukoliko korisnik ipak ne želi mjesto u tome redu može ga otkazati.
- Korisnik je krivo unio podatke, te ode na uređivanje korisničkih podataka
- Šalterski službenik nakon što završi rad sa klijentom, pozove sljedećeg klijenta tako da promijeni aktivan broj



Slika 1: Use case dijagram sustava

2 REST WEB SERVISI

Web servis je skup otvorenih protokola i standarda koji se koriste za razmjenu podataka između aplikacija ili sustava. Softverske aplikacije koje su pisane u različitim programskim jezicima i koje se izvode na različitim platformama mogu koristiti web servise za razmjenu podataka preko računalnih mreža poput Interneta na način sličan interprocesnoj komunikaciji (*IPC*) na jednom računalu. Ova interoperabilnost (npr. između Java i Python ili Windows i Linux aplikacija) rezultat je upotrebe otvorenih standarda. (Tutorials Point (I) Pvt. Ltd., 2018)

Web servis za upravljanje redovima čekanja bi trebao biti u skladu sa REST konvencijom. To omogućuje da krajnje točke servisa, odnosno *API*[1], budu po standardu gdje kad određena aplikacija ili web stranica pokušava pristupiti *API*-ju servisa zna definirati zahtjev i kakav će dobiti odgovor od servisa.

REST (*engl. Representational State Transfer*) je arhitektonski stil koji definira ograničenja kao što su homogena sučelja, koja primjenom na web servise potiče željena svojstva, od performansi servisa pa sve do skalabilnosti i promjenjivosti sustava što omogućuje da servisi rade najbolje preko Web-a. (Oracle, 2013)

U tom stilu, podaci i funkcionalnosti se smatraju resursima te se pristupaju preko URI-a (*engl. Uniform Resource Identifiers*), odnosno linkovi na Webu. Resursi se definiraju pomoću skupa jednostavnih ali dobro definiranih operacija. REST arhitekatski stil ograničava arhitekturu na klijent – server te je dizajniran da koristi komunikacijski protokol bez stanja gdje je to najčešće HTTP protokol. U tom stilu klijenti – serveri razmjenjuju zastupljenost resursa korištenjem standardiziranog sučelja i protokola.

Sljedeći principi podupiru REST aplikacije da budu jednostavniji, lagani i brzi:

(Oracle, 2013)

- Identifikacija resursa preko URI-a: REST servis izlaže skup resursa koji identificira ciljeve interakcije sa klijentom (*engl. endpoints*). Resursi se identificiraju preko URI-ja koji pružaju globalni adresni prostor za resurse i otkriće servisa.

[1] Aplikacijsko programsko sučelje (engl. Application programming interface)

- Uniformno sučelje: Resursi se upravljaju korištenjem fiksnog seta četiri vrste operacija – PUT, GET, POST i DELETE. PUT stvara novi resurs koji se zatim može izbrisati korištenjem DELETE. GET dohvaća trenutačno stanje resursa u dogovorenom obliku prikaza, dok POST prenosi novo stanje u resurs.
- Samo opisne poruke: Resursi su odvojeni od prikaza tako da bi se njihov sadržaj mogao pristupiti u različitim formatima kao što su HTML, XML, TXT, PDF, JPG, JSON i drugi. Meta podaci o resursu su dostupni te se koriste za, npr. kontroliranje predmemorije podataka, detekciju prijenosa grešaka, pregovaranje oko formata prikaza podataka, izvršavanje autentikacije ili kontrolu pristupa.
- Interakcije sa stanjima preko hiperveza: Svaka interakcija sa resursom je bez stanja; zahtjevi su samostalni. Interakcije sa stanjima se temelje na konceptu eksplicitnog stanja transakcije. Postoje nekoliko tehnika za izmjenu stanja, kao što su prepisivanje URI-a, kolačići i polja sakrivenih formi. Stanje se može ugraditi u poruku odgovora da bi se moglo pokazati na važeća buduća stanja interakcije.

3 ODABRANE TEHNOLOGIJE ZA MREŽNU USLUGU

U ovom poglavlju će se istražiti potrebne tehnologije za izradu web servisa upravljanja redovima čekanja. Cilj je izrada te mrežne usluge, odnosno web servisa sa osnovnim funkcionalnostima. Za ostvarenje toga cilja potrebno je istražiti potrebne tehnologije koje bi bile najbolje i najjednostavnije za izradu takvog servisa. Programski jezik je Java stoga će se istraživati tehnologije vezane za to područje.

3.1 SPRING BOOT

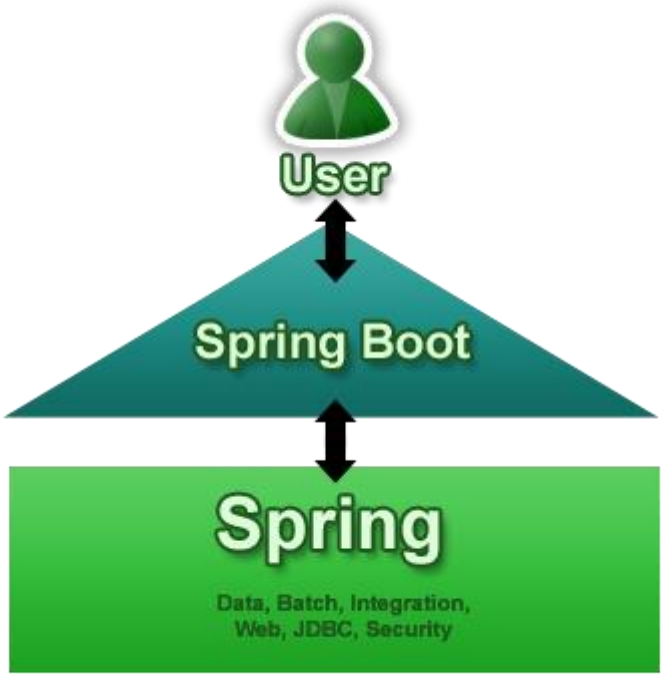
Web servis za upravljanje redovima čekanja pod nazivom „JustInTime“ je naslijeđeni projekt sa kolegija „Izrada informatičkih projekata“ u kojem se je već tada odlučilo za „Spring Boot“ iz razloga što je najpoznatije i najhvaljenije programsko radno okruženje za izradu REST servisa u Java programskom jeziku. Osim toga, velika većina IT poduzeća ga koristi u produkciji budući da ima najveću zajednicu, podršku te stabilnost. Budući da kvalitetne alternative „Spring Boot“-a sa povećom zajednicom i podrškom baš i nema nadalje će se istražiti više o njemu.

„Spring Boot“ je programski okvir koje omogućuje izradu produkcijskih „Spring“ aplikacija i servisa jednostavno i bez velikoga napora. Sa njime se mogu puno agilnije razvijati „Spring“ aplikacije te se više može fokusirati na funkcionalnosti aplikacije sa minimalno razmišljanja o konfiguraciji „Spring“-a. Pomno izabrane značajke „Spring“ platforme omogućavaju intuitivno korištenje i izradu naprednih aplikacija.

„Spring“ je započeo kao jednostavna i lagana alternativa „Java Enterprise Edition“-u. Umjesto razvoja glomaznih komponenti kao što su „Enterprise JavaBeans“ (EJBs), „Spring“ je ponudio jednostavniji pristup poslovnom razvoju Java aplikacija koristeći ubrizgavanje ovisnosti i programiranje usmjereno na aspekte kako bi se postigle mogućnosti od EJB-a sa običnim Java objektima. Ali, dok je „Spring“ jednostavan i lagan za razvoj poslovnih komponenata, toliko je težak za konfigurirati. Vrijeme utrošeno na pisanje konfiguracija znači vrijeme koje se moglo iskoristiti u pisanju aplikacijske logike. Osim toga, upravljanje ovisnošću je još jedan od važnih dijelova projekta koji je uvijek potreban. Kada se dodaje zavisnost u projektu ne piše se kod aplikacije već se dodaje biblioteka. Bilo koja nekompatibilnost koja proizlazi iz

odabira pogrešnih verzija ovisnosti može uvelike smanjiti produktivnost. „Spring Boot“ je to sve promijenio. (Walls, 2016)

Dijagram ispod predstavlja „Spring Boot“ kao točku fokusa većeg „Spring“ ekosustava. Prikazuje manji dio cjeline gdje korisnici mogu iskoristiti vrijednosti preostalog dijela „Spring“-a:



Slika 2: Spring Boot kao dio Spring ekosustava (Webb, 2013)

Primarni ciljevi „Spring Boot“-a su sljedeći: (Webb, 2013)

- Pružanje radikalno bržeg početnog iskustva i široku pristupačnost prilikom svih „Spring“ razvijanja.
- Osjećaj samouvjerenosti od samog početka korištenja te prilagodljivost i fleksibilnost kako se zahtjevi počinju granati od početnih
- Pružanje velikog broja ne funkcionalnih značajka koji su dio velikog broja projekata (npr. ugrađeni poslužitelji, zaštita, metrike, eksterna konfiguracija)

„Spring Boot“ ne generira kod i nema nikakve potrebe za XML konfiguracijama.

„Spring Boot“ ima naredbenu aplikaciju koja omogućuje da se izvršavaju „Spring“ skripte. Skripte se pišu u „Groovy“ programskom jeziku, što znači da je

sintaksa slična kao i u Javi samo sa čišćim kodom bez potrebnog ponavljajućeg dijela koda (*engl. Boilerplate code*[2]). Moguće je utvrditi puno informacija samo čitajući kod skripte. (Webb, 2013)

Kada se pokrene aplikacija naredbom *spring run* događa se nekoliko stvari: (Webb, 2013)

- Skripta sadrži najčešće „import“ naredbe čime se skraćuje potreba za nepotrebnim tipkanjem
- *@ResponseBody* anotacija omogućuje da se automatski pripreme potrebne „Spring“ *JAR* datoteke
- Automatski se kreira anotacija *@Configuration* koje bi inače trebalo dodatno ručno napisati
- Pokrene se ugrađeni server koji upravlja dolaznim zahtjevima na portu 8080

Anotacija *@SpringBootApplication* koja se nalazi na početnoj (*main*) klasi aplikacije omogućuje skeniranje komponenti „Spring“-a i auto konfiguraciju „Spring Boot“-a. Zapravo, *@SpringBootApplication* kombinira još tri korisne anotacije: (Walls, 2016)

- „Spring“-ov *@Configuration* – Određuje klasu kao konfiguracijsku klasu pomoću „Spring“ Java konfiguracije.
- „Spring“-ov *@ComponentScan* – Omogućuje skeniranje komponenata tako da će web kontroler klase i druge komponente biti automatski otkrivene i registrirane kao *bean*-ovi u aplikacijskom kontekstu „Spring“-a.
- „Spring Boot“-ov *@EnableAutoConfiguration* – Anotacija koja čini srž „Spring Boot“-a, a to jedna linija konfiguracije koja omogućuje cijelu čaroliju auto konfiguracije. Anotacija uvelike smanjuje pisanje koda konfiguracije jer bez nje bi se konfiguracije morale ručno napisati.

„Spring Boot“ donosi mnogo magije u razvoju „Spring“ aplikacija. Ali postoje četiri osnovna trika koje izvodi: (Walls, 2016)

[2] *Boilerplate code* se odnosi se na dijelove koda koji se moraju uključiti u mnoga mjesta sa malo ili bez ikakvih promjena. Pisanje puno koda za minimalno posla.

- **Automatska konfiguracija** – „Spring Boot“ može automatski pružiti konfiguraciju za funkcionalnosti aplikacije koja je zajednička mnogim „Spring“ aplikacijama.
- **Starter zavisnosti** (*engl. Starter dependencies*) – „Spring Boot“-u se definira koje su funkcionalnost potrebne za aplikaciju gdje će se pobrinuti i osigurati da se potrebne biblioteke dodaju u izgradnju aplikacije.
- **Sučelje naredbenog retka** – Ova dodatna značajka „Spring Boot“-a omogućuje pisanje kompletne aplikacije od čistog koda aplikacije gdje više nema potrebe za tradicionalnom izgradnjom projekta.
- **Aktivator** – alat koji daje uvid u detalje pokrenute „Spring Boot“ aplikacije.

3.1.1 Automatska konfiguracija

U bilo kojem „Spring“ aplikacijskom kodu može se pronaći Java konfiguracija ili XML konfiguracija (ili oboje) koje omogućuju određene pomoćne značajke i funkcionalnosti za aplikaciju. Na primjer, kod aplikacije koja pristupa relacijskoj bazi podataka sa JDBC[3]-om se mora konfigurirati „Spring“-ov *JdbcTemplate* kao „Spring“ *bean* u kontekstu aplikacije. Ova vrlo jednostavna deklaracija *bean-a* stvara instancu *JdbcTemplate* koja se injektira sa svojom ovisnošću – u ovom slučaju je to *DataSource*. Naravno, to znači da se također treba konfigurirati *DataSource bean* tako da ovisnost može biti ispunjena. (Walls, 2016)

„Spring Boot“ može automatski konfigurirati ove uobičajene scenarije konfiguracije. Ako „Spring“ za vrijeme podizanja sustava otkrije da ima biblioteku baze podataka u putanji klase (*engl. classpath*) aplikacije, automatski će konfigurirati ugrađenu bazu podataka. Ako je uključen *JdbcTemplate classpath*, onda će također konfigurirati *JdbcTemplate bean*. Nema potrebe da se vodi briga o konfiguriranju tih *bean*-ova. Oni će biti spremni za ubrizgavanje u bilo koji *bean* koji se koristi. (Walls, 2016)

[3] Java Database Connectivity – driver koji služi za spajanje na bazu i izvršavanje SQL naredbi

3.1.2 Starter zavisnosti

„Spring Boot“ nudi pomoć pri upravljanju ovisnošću u projektu putem startera zavisnosti. Zavisnosti startera zapravo su samo posebne zavisnosti „Maven[4]“ ili „Gradle“ koji iskorištavaju prijelazne zavisnosti da agregira najčešće korištene biblioteke pod zavisnosti značajki. Na primjer, za REST API sa „Spring MVC“-om koji ima *JSON* resurse te deklarativnu provjeru valjanosti po specifikaciji *JSR-303* i posluživanje aplikacije pomoću ugrađenog *Tomcat* servera potrebno je najmanje osam zavisnosti da bi se projekt mogao sagraditi: (Walls, 2016)

- *org.springframework:spring-core*
- *org.springframework:spring-web*
- *org.springframework:spring-webmvc*
- *com.fasterxml.jackson.core:jackson-databind*
- *org.hibernate:hibernate-validator*
- *org.apache.tomcat.embed:tomcat-embed-core*
- *org.apache.tomcat.embed:tomcat-embed-el*
- *org.apache.tomcat.embed:tomcat-embed-logging-jul*

S druge strane, korištenjem „Spring Boot“ starter zavisnosti dovoljno je dodati *web starter* (*org.springframework.boot:spring-boot-starter-web*) kao zavisnost izgradnje gdje će ta jedna zavisnost privremeno povući sve druge potrebne zavisnosti tako da se ne moraju ručno pretraživati. Osim toga bitna stvar za naglasiti je što dodavanjem određene ključne riječi, kao što je u ovom primjeru to „web“, definira tip funkcionalnosti za što će se aplikacija koristiti. Tako da za *JPA* repozitorije dodaje se ključna riječ „jpa“, dok za funkcionalnosti za sigurnost se dodaje „security“. Također, starter zavisnosti „Spring Boot“-a rješava problem zabrinutosti oko toga koje verzije biblioteke je potrebno imati u projektu. Verzije biblioteka koje starter povuče su sve testirane zajedno tako da se sa sigurnošću neće dogoditi nekompatibilnosti među njima. (Walls, 2016)

[4] Maven i Gradle su alati za upravljanje izgradnjom projekta i zavisnih biblioteka

3.1.3 Sučelje naredbenoga retka

Sučelje naredbenoga retka (CLI) „Spring Boot“-a iskorištava starter zavisnosti i auto konfiguraciju kako bi se razvojni programer mogao usredotočiti na pisanje aplikacijskoga koda. CLI detektira koje vrste zavisnosti se koriste i zna koje starter zavisnosti treba dodati na putanju klase kako bi funkcioniralo. Jednom kada su te zavisnosti na putanji klase, starta se niz automatske konfiguracije koji osigurava da su *DispatcherServlet* i „Spring“ *MVC* omogućeni tako da kontroler može odgovarati na HTTP zahtjeve. CLI je izborni dio „Spring Boot“ asortimana. Iako pruža ogromnu snagu i jednostavnost za razvoj „Spring“ aplikacija također uvodi i prilično nekonvencionalni model razvoja tako da se po izboru može zaobići. (Walls, 2016)

3.1.4 Aktivator

Aktivator nudi pregled rada aplikacije sa pojedinostima kao što su *bean*[5]-ovi koji su konfigurirani u aplikacijskom „Spring“ kontekstu, definirane postavke od strane auto konfiguracije „Spring Boot“-a, koje su varijable okruženja, svojstva sustava, svojstva konfiguracije i argumenti naredbenog retka dostupni aplikaciji koja je pokrenuta. Osim toga, prikazuje trenutačno stanje dretvi koje su u pokrenutoj aplikaciji, stanje nedavnih *HTTP* zahtjeva kojima je aplikacija upravljala, razne metrike za korištenje memorije, sakupljača smeća, web zahtjeve i korištenja izvora podataka. Aktivator izlaže ove informacije na dva načina: preko krajnje točke weba ili preko sučelja ljuske. U potonjem slučaju se zapravo može otvoriti sigurna ljuska (*SSH*) na aplikaciju te izdati naredbe za praćenje zahtjeva dok se aplikacija pokreće. (Walls, 2016)

[5] Objekti koji su instancirani, sastavljeni i upravljani od strane „Spring“ *Inversion of Control* spremnika.

3.2 BAZA PODATAKA – SQL I NOSQL

Na web servis za upravljanje redovima čekanja „JustInTime“, kao i kod većine servisa koji rade sa podacima očekuju određenu bazu podataka, može se primijeniti bilo koja baza podataka. Opcije su bile korištenje određene relacijske SQL baze podataka ili neke od NoSQL baza podataka. U nastavku rada je istraživanje o prednostima i nedostacima obje baze podataka te u konačnici koji je najbolji izbor za „JustInTime“.

Kada se spomene baza podataka najčešće se misli na konvencionalnu SQL relacijsku bazu podataka budući da je to proizvod tehnološke evolucije, dobre prakse te korištenja u produkcijskim stresnim uvjetima. Dizajnirane su za pouzdane transakcije i ad hoc upite koje su ključne točke poslovnih aplikacija, ali također dolaze i sa mnoštvo ograničenja kao što su stroge sheme što ih čini manje prikladnim za druge vrste aplikacija. Poznato je da SQL baze podataka imaju inherentnu strukturu. Konvencionalne baze podataka kao što su „Microsoft SQL Server“, „MySQL“, ili „Oracle Database“ koriste strogu shemu koja je formalna definicija kako će podaci biti umetnuti u bazu podataka. Na primjer, određeni stupac u tablici može biti ograničen samo na tip podataka „INTEGER“. Kao rezultat toga, podaci snimljeni u stupcu imat će visoki stupanj normalizacije. Stroga shema SQL baze podataka također čini relativno lako za izvesti agregaciju podataka, na primjer preko naredbe „JOIN“. (Yegulalp, 2017)

NoSQL baze podataka su nastale kao odgovor na stroga ograničenja SQL baza podataka. NoSQL sustavi pohranjuju i upravljaju podacima na način koji omogućava veliku operativnu brzinu i veliku fleksibilnost od strane programera. Razvitku su mnogo pridonijele tvrtke kao što su Google, Amazon, Yahoo i Facebook koje su tražile bolje načine pohrane sadržaja ili procesnih podataka za masivne web stranice. Za razliku od SQL baze podataka, mnoge NoSQL baze podataka mogu se skalirati na preko stotinu ili više tisuća servera.

Prednosti NoSQL-a ipak ne dolaze bez troškova. NoSQL sustavi obično ne daju istu razinu dosljednosti podataka kao i SQL baze podataka. Zapravo, iako SQL baze podataka tradicionalno žrtvuju performanse i skalabilnost za svojstva ACID-a (*engl. Atomicity, Consistency, Isolation, Durability*) iza pouzdanih transakcija, NoSQL baze

podataka u velikoj su mjeri odbacile ta svojstva ACID-a da bi dobile na brzini i skalabilnosti. (Yegulalp, 2017)

U NoSQL-u, podaci se mogu pohraniti bez sheme ili u slobodnom obliku. Svi se podaci mogu pohraniti u bilo koji zapis. Među NoSQL bazama podataka postoje četiri uobičajena modela za pohranu podataka, što dovodi do četiri uobičajene vrste NoSQL sustava: (Yegulalp, 2017)

1. Baza podataka dokumenata (npr. „CouchDB“, „MongoDB“). Umetnuti podaci pohranjeni su u obliku slobodnih JSON struktura ili "dokumenata", gdje podaci mogu biti bilo što – od cijelih brojeva do slobodnog teksta. Nema inherentne potrebe da se navedu polja unaprijed, jer će ih NoSQL dodati u onom trenutku kada bude bilo potrebno da se nadoda u dokument.
2. Pohrane u obliku ključ – vrijednost (npr. „Redis“, „Riak“). Vrijednosti u obliku slobodnih podataka – od jednostavnih cijelih brojeva ili niza znakova do kompleksnih JSON dokumenata koji se pristupaju pomoću ključeva.
3. Pohrane u obliku širokih stupaca (npr. „HBase“, „Cassandra“). Podaci se pohranjuju u stupce umjesto redaka kao u uobičajenom SQL sustavu. Bilo koji broj stupaca (a time i mnogo različitih vrsta podataka) se mogu grupirati ili grupirati po potrebi za upite odnosno prikaze podataka.
4. Graf baze podataka (npr. „Neo4j“). Podaci se predstavljaju kao mreža ili graf entiteta i njihovih odnosa gdje svaki čvor na grafu ima slobodni oblik podataka.

NoSQL baza podataka je korisna u sljedećim situacijama: (Yegulalp, 2017)

1. Brzi pristup podacima gdje je bitniji naglasak na brzinu i jednostavnost pristupa od pouzdanih transakcija ili konzistentnosti podataka.
2. Pohrana velike količine podataka neovisno o shemi gdje bi kod konvencionalnih baza podataka taj proces bio težak i spor.
3. Korištenje nestrukturiranih podataka iz različitih izvora te sposobnost spremanja istih u izvornom obliku što nudi maksimalnu fleksibilnost.
4. Mogućnost pohranjivanja podataka u hijerarhijsku strukturu gdje podaci opisuju sami sebe bez potrebe vanjskih shema za objašnjenje podataka.

Podaci mogu biti u slobodnom obliku te detaljno opisani gdje dodatno tumačenje nije potrebno.

Najčešće korištene NoSQL dokument baze podataka su „CouchDB“ i „MongoDB“ gdje „MongoDB“ zauzima prvo mjesto. Budući da su oba sustava na vrhu ljestvice rangiranja provjeriti će se njihove prednosti i mane te koji sustav je bolje primjenjiv za određenu poslovnu logiku. (solid IT, 2018)

3.2.1 CouchDB

CouchDB je napisan u „Erlang[6]“-u i dostupan je za Android, BSD, iOS, Linux, OS X, Solaris i Windows operativne sustave. Izvorno ga je izradio Damien Katz 2005., bivši IBM-ov razvojni programer koji je radio na „Lotus Notes“.

CouchDB koristi pohranu dokumenata s podacima koji se prikazuju u JSON formatu. Nudi precizan HTTP API za čitanje, dodavanje, uređivanje i brisanje dokumenata baze podataka. Svaki dokument sastoji se od polja i privitaka. Polja se mogu sastojati od brojeva, teksta, logičkih izraza, popisa i još mnogo toga. Model ažuriranja za CouchDB je optimiziran i ne zaključava preostale akcije na bazi podataka. Struktura baze podataka je inspirirana „Lotus Notes“-om i nudi mogućnost skaliranja od globalnih klastera do mobilnih uređaja. Primjeri tvrtki koji ga koriste u redovnom poslovanju: „Talend SA“, „Akamai Technologies“, „Hothead Games Inc.“, „GenCorp Technologies“, „Vivint Solar Inc.“. Trenutno CouchDB-om upravlja tvrtka „Apache Software Foundation“. (Sarig, 2017)

Prikazi u CouchDB su slični indeksima u SQL. Oni se mogu koristiti za filtriranje dokumenata, dohvaćanje podataka u određenoj narudžbi i stvaranje učinkovitih indeksa za pronalazak dokumenta pomoću vrijednosti sadržanih u tim dokumentima. Takvi indeksi mogu predstavljati odnose između dokumenata. Rezultati prikaza pohranjeni su u vrsti strukture indeksa vrste B-stabla.

CouchDB koristi „MapReduce[7]“, proces u dva koraka koji gleda sve dokumente i stvara mapu sa rezultatima koji se sastoji od uređene liste parova ključ – vrijednost.

[6] Funkcijski programski jezik razvijen od strane Ericsson-a
[7] Algoritam i programski model razvijen od strane Google-a

Mapiranje se događa jednom nakon stvaranja dokumenta. Nakon toga se ne mijenja te se promjeni isključivo samo ako se dogodi ažuriranje dokumenta. (Sarig, 2017)

Podržava i „master-master“ i „master-slave“ replikaciju. To omogućuje niske latencije pristupa podacima bez obzira na lokaciju. Replikacija je vrlo jednostavna te se svodi na slanje HTTP zahtjeva u bazu podataka sa željenim izvorom na definirani cilj. Promjene, koje se nalaze u izvoru HTTP zahtjeva, će se početi slati u odabranu bazu podataka. Takav proces je jednosmjeran. Za dvosmjerni proces potrebno je pokrenuti replikaciju na odredišnom serveru kao izvor te na udaljenom server kao odredište. (Sarig, 2017)

Mobilna podrška: Prednost mu je što se može prikazivati na mobilnim uređajima „Android“ ili „iOS“. Osim toga baza podataka se može sinkronizirati s udaljenom glavnom bazom podataka čime se podaci lako dijele između mobilnih uređaja i servera. (Sarig, 2017)

Snimke: Sve promjene u dokumentu pojavljuju se kao izmjene te se dodaju informacije u datoteku. To znači da se može uhvatiti snimka (*engl. snapshot*) datoteke i kopirati ga na drugo mjesto pa čak dok je baza podataka pokrenuta bez brige da bi moglo doći do korupcije podataka. (Sarig, 2017)

Replikacija: Nudi mogućnost replikacije „master-master“ i „master-slave“. Budući da CouchDB čini samo izmjene na bazi podataka smanjuje se rizik od sukoba podataka. Osim toga nudi mogućnost replikacije „master-master“ gdje se svi serveri dvosmjerno repliciraju. Tu je i mogućnost selektivne replikacije gdje se definiranim filterima mogu kontrolirati koji će se dokumenti kopirati na uređaj. To omogućuje mobilnom uređaju sa manjom memorijom da ima podskup baze podataka. (Sarig, 2017)

Upiti: CouchDB upiti upotrebljavaju „MapReduce“ funkcije čiji koncept može biti težak za shvatiti kod pojedinaca naučenih na SQL sintaksu. Međutim s malo truda i vremena moguće je pronaći brzo i elegantno rješenje. (Sarig, 2017)

3.2.2 MongoDB

MongoDB je napisan u C++ i može se koristiti na Linux, OS X, Solaris i Windows operativnim sustavima. Početnu verzija je napravila tvrtka 10gen 2007. koja je kasnije promijenila ime u MongoDB Inc. i današnji je vlasnik projekta.

MongoDB pohranjuje podatke bez sheme pomoću dokumenata u BSON formatu. Ove zbirke dokumenata ne moraju imati unaprijed definiranu strukturu, a stupci mogu varirati za različite dokumente u zbirci. Pohrana podataka bez sheme omogućuje stvaranje dokumenata bez prethodnog definiranja strukture za taj dokument. Istodobno ima mnoge značajke relacijskih baze podataka, uključujući snažnu konzistentnost podataka i izražajni jezik upita. (Sarig, 2017)

U MongoDB-u poželjno je imati indekse zbog ubrzanja upita jer se osjeti znatna razlika u vremenu čitanja indeksiranih i ne indeksiranih podataka.

MongoDB nudi „single-master“ replikaciju sa ugrađenim automatskim izborima. Promovira sekundarnu bazu podataka (auto-izbor) ako primarna baza podataka postane nedostupna. Uz setove replika, može postojati jedna primarna baza podataka s više repliciranih baza podataka koje imaju sekundarnu ulogu. (Sarig, 2017)

Upiti: MongoDB je bliži SQL-u, što znači da će SQL korisnicima biti lakše za shvatiti i naučiti.

Brzo čitanje: MongoDB pruža brže čitanje od CouchDB-a jer koristi binarni protokol koji je brži od CouchDB-ove REST HTTP API metode.

Replikacija: MongoDB nudi samo replikaciju „master-slave“ preko setova replikacije. Prema tvrdnji Riyad Kalla, direktora globalne kreditne platforme na „PayPal“-u, za korištenje više „master“-a u Mongo okruženju potrebno je postaviti „sharding“ uz replike setove gdje će svaki „shard“ biti replika seta sa mogućnošću pisanja svakom „master“-u u svakom setu. Nažalost, to dovodi do mnogo složenijih postavki gdje ne može svaki poslužitelj imati punu kopiju skupa podataka. (Sarig, 2017)

Veličina: MongoDB je najbolji izbor za pohranu velike količine podataka te promjenjivim i rastućim setom podataka.

3.2.3 Usporedba CouchDB i MongoDB

Sljedeća tablica prikazuje usporedbu baza podataka CouchDB i MongoDB.

	CouchDB	MongoDB
Brzina	Ako je brzina kritična, MongoDB je brži	Nudi brže čitanje podataka
Mobilna podrška	iOS i Android operacijski sustavi	Nema mobilnu podršku
Replikacija	„master-master“ i „master-slave“ replikacija	„master-slave“ replikacija
Veličina	Podržava određeni rast baze podataka, dok MongoDB ima bolju podršku za brži rast i kad struktura podataka nije definirana	Najbolja opcija za ne strukturirane podatke i brzo rastuće podatke
Sintaksa	Upiti preko „MapReduce“ funkcija, teže za shvatiti korisnicima SQL-a	Lakše za shvatiti korisnicima SQL-a zbog slične sintakse

Tablica 1: Usporedba CouchDB i MongoDB (Sarig, 2017)

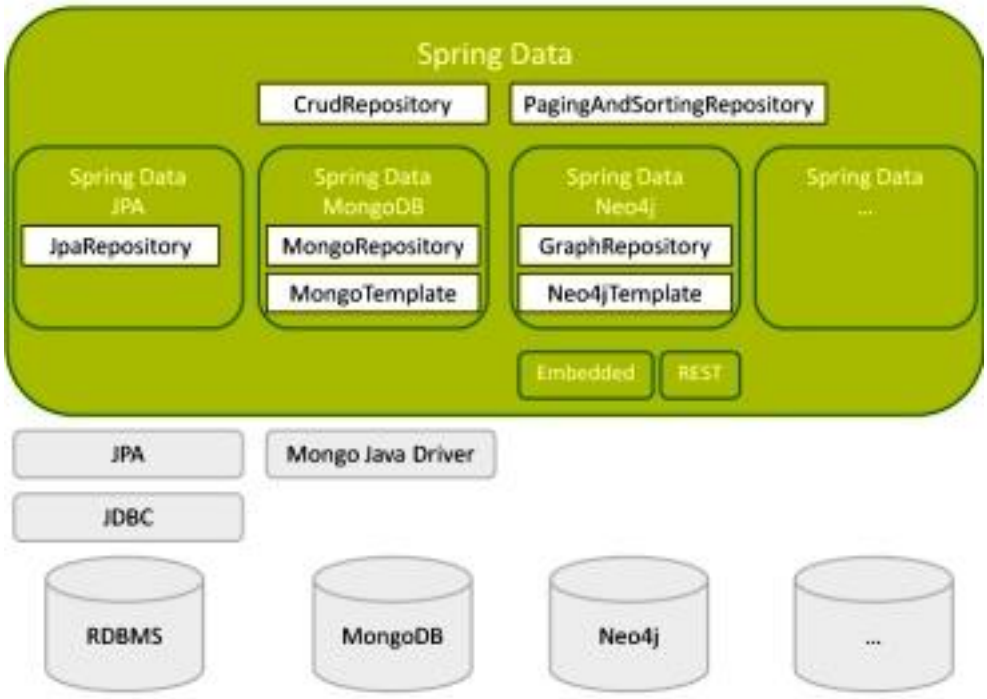
CouchDB je odličan izbor ukoliko je potrebno imati bazu podataka koja radi na mobilnim uređajima, ima podršku za „,master-master“ replikaciju ili izdržljivost jednog servera. Ako su potrebne maksimalne performanse ili je stalno rastuća baza podataka onda je MongoDB bolji izbor.

Prilično je jasno iz priloženih informacija koja bi baza podataka najbolje odgovarala web servisu za upravljanje redovima čekanja „JustInTime“ te shodno tome koristit će se NoSQL MongoDB baza podataka.

3.3 SPRING DATA

Budući da je izabran „Spring Boot“ kao programski okvir najsmislenije je da se koriste njegove specifične biblioteke za daljnji rad sa podacima servisa za upravljanje redovima čekanja. Potrebna tehnologija za rad sa bazama podataka i entitetima je „Spring Data“ koja ima mogućnost znatnog olakšavanja posla tako da se izbjegne pisanje SQL upita na bazu podataka te automatskog povezivanja entiteta modela u servisu sa onima u bazi podataka. U daljnjem istraživanju će se dobiti više informacija o tome.

„Spring Data“ je jedan od vodećih projekta „SpringSource[8]“-a čija je svrha unificirati i olakšati pristup različitim vrstama pohranjivanja podataka od relacijskih do NoSQL baza podataka.



Slika 3: Shema povezanosti Spring Data sa bazama podataka (Trelle, 2012)

Uz svaku vrstu pohrane podataka, repozitoriji (npr. *DAO*-ovi ili Objekti za pristup podacima) obično nude CRUD (*Create-Read-Update-Delete*) operacije na pojedinačnim domenskim objektima, metodama traženja, sortiranja i brojanja. „Spring

[8] Tvrtka koja je osnovala Spring Framework

Data“ pruža generička sučelja za ove aspekte (C*rudRepository, PagingAndSortingRepository*), kao i specifične implementacije pohrane podataka.

„Spring“ objekti predložaka (kao što je „JdbcTemplate“) služe da bi se mogle napisati prilagođene implementacije repozitorija. Dok su objekti predložaka moćni, postoji i bolje rješenje. Pomoću „Spring Data“ repozitorija potrebno je samo napisati sučelje pomoću metoda za pretraživanje definiranih prema vrsti pohrane podataka koje se koriste te određenom skupu konvencija. „Spring Data“ će pružiti odgovarajuću implementaciju tog sučelja tijekom izvođenja aplikacije. Na primjer, slika ispod prikazuje kako definirati sučelje te metode koje će se koristiti za pretraživanje baze podataka. (Trelle, 2012)

```
interface UserRepository extends MongoRepository<User, String> {
    @Query("{ fullName: ?0 }")
    List<User> findByTheUsersFullName(String fullName);

    List<User> findByFullNameLike(String fullName, Sort sort);
}
```

Slika 4: Primjer korištenja Spring Data sa MongoDB. (Trelle, 2012)

JPA je uveo standard za objektno-relacijsko mapiranje (*engl. ORM – Object Relational Mapping*), odnosno povezivanje aplikacijskih objekata sa podacima u tablicama baza podataka. „Hibernate“ je najčešći *ORM* programski okvir koji implementira JPA specifikaciju. Glavna mu je uloga mapiranja Java objekta sa tablicama baza podataka te mapiranje Java tipova podataka za SQL tipovima podataka. Također, generira SQL pozive preko definiranog drivera direktno na bazu podataka.

Sa „Spring Data“-om ta je podrška proširena na *NoSQL* baze podataka sa objektnim strukturama podataka. Ali te strukture podataka mogu biti poprilično međusobno različite pa bi bilo teško izraditi zajednički API za mapiranje objekata i podataka. Svaka vrsta repozitorija baze podataka dolazi sa vlastitim skupom anotacija za pružanje potrebnih meta informacija za mapiranje objekata. U tablici ispod su jednostavni primjeri korisničkog objekta koji se mapira sa podatkovnim datotekama u različitim bazama podataka: (Trelle, 2012)

JPA[9]	MongoDB	Neo4j[10]

@Entity
@Table(name="TUSR")
public class User {
  @Id
  private String id;
  @Column(name="fn")
  private String name;
  private Date lastLogin;
}
	@Document(
	collection="usr")
	public class User {
	  @Id
	  private String id;
	  @Field("fn")
	  private String name;
	  private Date lastLogin;
	}
		@NodeEntity
		public class User {
		  @GraphId
		  Long id;
		  private String name;
		  private Date lastLogin;
		}

Tablica 2: Primjeri entiteta različitih baza podataka (Trelle, 2012)

„Spring Data“ za relacijske baze podataka koristi *JPA* anotacije entiteta. Mapiranje se provodi u *JPA* implementaciji koji je definiran u projektu gdje je to najčešće „Hibernate“ za relacijske baze podataka. „MongoDB“ i „Neo4j“ zahtijevaju sličan skup anotacija. Definira se na razini klase kako bi se mapirala klasa sa kolekcijom („MongoDB“ koristi kolekcije dokumenata za spremanje podataka) ili tipu čvora grafa baze podataka „Neo4j“ (čvorovi i rubovi su glavne vrste podataka u graf bazama podataka). (Trelle, 2012)

Svaki *JPA* entitet mora imati jedinstveni identifikator. Tako vrijedi i za „MongoDB“ dokumente i „Neo4j“ čvorove. „MongoDB“ koristi *@Id* anotaciju dok „Neo4j“ koristi *@GraphId* anotaciju za definiranje jedinstvenog identifikatora. Vrijednosti ovih atributa se popunjuju nakon što se entitet sačuva u bazi podataka. Za ostale atribute entiteta se može upotrijebiti anotacija *@Field* ako naziv atributa u „MongoDB“ dokumentu ne odgovara atributu Java entiteta. (Trelle, 2012)

[9] JPA – Java Persistance API se koristi sa relacijskim bazama podataka
[10] Neo4j – graf baza podataka

Također su podržane i reference na druge objekte. Na primjer, uloga korisnika može se sačuvati na ovaj način:

JPA	MongoDB	Neo4j
@OneToMany private List<Role> roles;	private List<Role> roles;	@RelatedTo(type = "has", direction = Direction.OUTGOING) private List<Role> roles;

Tablica 3: Primjeri referenca u bazi podataka na objekte (Trelle, 2012)

JPA upotrebljava relaciju *@OneToMany*, što znači da je n strana spremljena u drugu tablicu te se obično upit izvodi pridruživanjem. „MongoDB“ ne podržava pridruživanje kolekcija dokumenata budući da se referentni objekti pohranjuju unutar istog dokumenta prema zadanim postavkama. Reference mogu biti i na druge dokumente što rezultiraju pridruživanju na strani klijenta. U graf bazi podataka „Neo4j“ relacije se nazivaju rubovi koji su jedan od osnovnih vrsta podataka takvih baza podataka.

„MongoDB“ i „Neo4j“ koriste mapiranje objekata koji je sličan poznatom *JPA ORM*-u, ali nije sasvim isti zbog različitih struktura podataka. Koncept koji stoji iza svih mapiranja je isti, što znači mapiranje Java objekata u strukture podataka koji su u pohranjeni u bazi podataka. (Trelle, 2012)

Najčešći je slučaj implementiranja CRUD operacija za pojedinačne zapise i metode pretraživanja entiteta. Metode pretraživanja imaju parametre koji se prosljeđuju u upit prije izvršavanja. Sa *JPA* pojavom CRUD operacije su postale dostupne putem *EntityManager* sučelja. Pisanje prilagođenih metoda pretraživanja i dalje je naporno budući da je potrebno definirati i izraditi upit, postaviti svaki parametar te izvršiti taj upit. Problem se može malo ublažiti korištenjem tečnog sučelja *TypedQuery,* ali je i dalje ipak potrebno implementirati metodu koja poziva metodu za postavljanje te izvršava upit za svaki definirani upit. Sa „Spring Data JPA“ JPQL[11] upiti se ne moraju deklarirati kao *@NamedQuerys* u klasi odgovarajućeg JPA entiteta.

[11] Java Persistence Query Language je neovisan o platformi objektno orijentirani upitni jezik baza podataka koji je dio JPA specifikacije

Umjesto toga, upit je anotacija metode repozitorija što se može vidjeti iz sljedećeg primjera: (Trelle, 2012)

```
@Transactional(timeout = 2, propagation = Propagation.REQUIRED)
@Query("SELECT u FROM User u WHERE u.name = 'User 3'")
List<User> findByGivenQuery();
```

Slika 5: Primjer korištenja @Query anotacije (Trelle, 2012)

Gore navedeno vrijedi i za „Spring Data“ „MongoDB“ i „Neo4j“. Naravno, konvencije o imenovanju metoda pretraživanja se međusobno razlikuju u vrsti baza podataka koje se koriste za pohranu podataka. Na primjer, „MongoDB“ podržava geoprostorne upite gdje se upiti mogu pisati na sljedeći način: (Trelle, 2012)

```
interface LocationRepository extends MongoRepository<Location, String> {

    List<Location> findByPositionWithin(Circle c);

    List<Location> findByPositionWithin(Box b);

    List<Location> findByPositionNear(Point p, Distance d);
}
```

Slika 6: Primjer korištenja geoprostornih upita u MongoDB (Trelle, 2012)

Također postoji i generička podrška za paginaciju i sortiranje tako da se dodijele posebni parametri za metode pretraživanja koji vrijede za sve vrste repozitorija baza podataka.

Glavne prednosti korištenja repozitorija su: (Trelle, 2012)

- Znatno je smanjeno pisanje *boilerplate* koda
- Upiti se mogu definirati zajedno sa metodom pretraživanja i dokumentacijom
- *JPQL* upiti se kompiliraju prilikom pokretanja „Spring“ konteksta a ne prilikom korištenja upita, što olakšava detektiranje grešaka sintakse

3.4 Spring Security OAuth2

Doći će vrijeme kada će biti potrebno određene krajnje točke sustava ograničiti određenim korisnicima te zaštititi sustav od neautoriziranog pozivanja *API*-ja. Budući da je kostur sustava „Spring“ programski okvir, za rad sa sigurnošću će se istražiti

moćna biblioteka „Spring Security“ sa OAuth2 standardom te prikazati potrebne komponente za njegovu implementaciju u servis upravljanja redovima čekanja.

U tradicionalnom modelu autentikacije klijent-server, klijent zahtijeva resurse sa ograničenjem pristupa (zaštićeni resurs) na serveru tako da se autenticira sa njim pomoću svojeg korisničkog imena i lozinke. Da bi pristupio aplikacijama treće strane sa ograničenim resursima vlasnik resursa, odnosno korisnik dijeli svoje korisničke podatke sa trećom stranom. Takav pristup otvara nekoliko problema te neophodna ograničenja: (Hardt, 2012)

- Aplikacije trećih strana moraju pohraniti tajne korisničke podatke vlasnika resursa za buduću upotrebu gdje obično lozinke budu u čistom tekst formatu.
- Serveri su dužni podržati provjeru autentičnosti pomoću lozinke unatoč tome što imaju inherentne sigurnosne slabosti.
- Aplikacije trećih strana zauzimaju pretjerano širok pristup zaštićenih resursa vlasnika ostavljajući ih bez ikakvih mogućnosti ograničavanja trajanja ili pristupa određenom ograničenom podskupu resursa.
- Vlasnici resursa ne mogu opozvati pristup nekoj trećoj strani bez povlačenja pristupa svim trećim stranama, a to mora učiniti tako da promjeni lozinku treće strane.
- Propust bilo koje aplikacije trećih strana dovodi do propusta lozinke krajnjeg korisnika te svih podataka zaštićenih lozinkom.

„OAuth“ rješava ove probleme uvođenjem sloja autorizacije i odvaja ulogu klijenta od vlasnika resursa. U „OAuth“ klijent zahtijeva pristup resursima kontroliranim od strane vlasnika resursa na serveru resursa te ima drukčije podatke od vlasnika resursa. Umjesto korištenja podataka vlasnika resursa za pristup zaštićenim resursima, klijent dobiva pristupni token (*engl. Access token*) što predstavlja niz koji označava određenu domenu, vijek trajanja i ostale atribute pristupa. Pristupni tokeni izdaju se klijentima treće strane od strane autorizacijskog servera sa odobrenjem vlasnika resursa. Klijenti koriste pristupni token kako bi pristupili zaštićenim resursima na serveru resursa. (Hardt, 2012)

„OAuth“ je dizajniran za upotrebu isključivo preko HTTP protokola te su ostali protokoli izvan njegove domene. Protokol „OAuth“ 1.0 rezultat je malih napora ad hoc

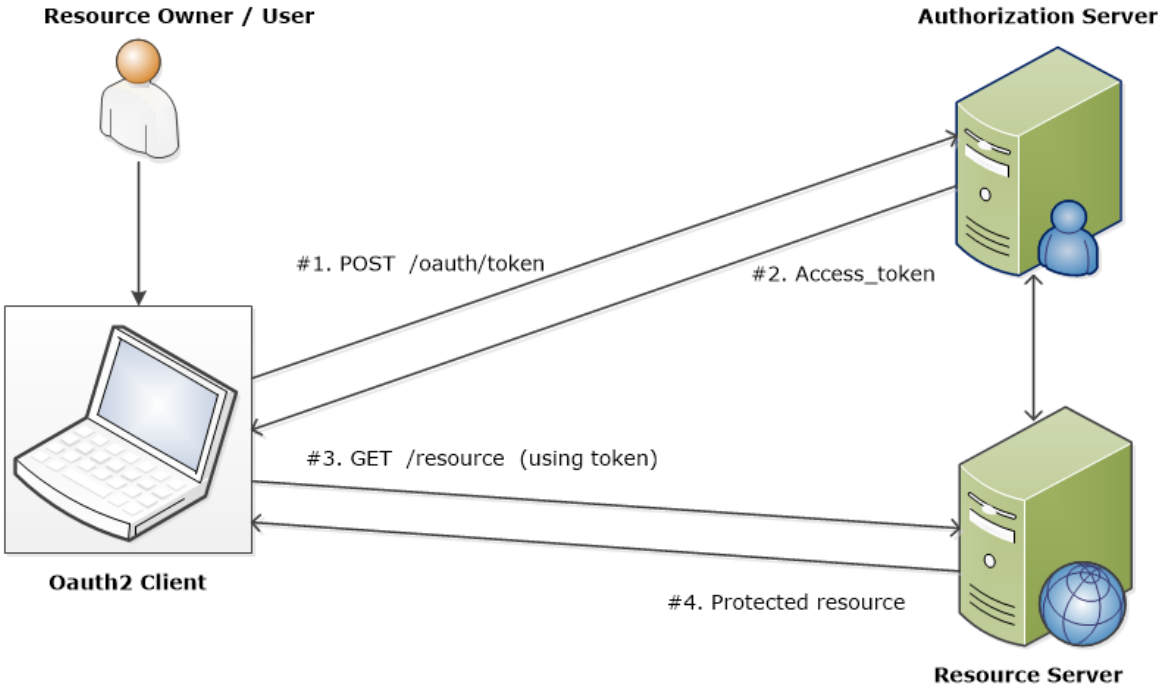
zajednice. „OAuth“ 2.0 protokol nije kompatibilan sa „OAuth“ 1.0. Obje verzije mogu postojati na mreži, dok implementacije mogu podržavati oboje. Međutim, namjera je da nove implementacije podržavaju „OAuth“ 2.0 te da se „OAuth“ 1.0 koristi samo za podršku kod postojećih implementacija. „OAuth“ 2.0 protokol dijeli vrlo malo implementacijskih pojedinosti sa „OAuth“ 1.0 protokolom. (Hardt, 2012)

3.4.1 Uloge

„OAuth“ definira četiri uloge:

- **Vlasnik resursa** – Entitet koji može dati pristup zaštićenom resursu. Kada je vlasnik resursa osoba onda se naziva krajnji korisnik.
- **Server resursa** – Server koji posjeduje zaštićene resurse sposobne za prihvaćanje i odgovaranje na zahtjeve zaštićenih resursa pomoću pristupnog tokena.
- **Klijent** – Aplikacija koja zahtjeva zaštićene resurse u ime vlasnika resursa sa njegovom autorizacijom. Pojam "klijent" ne podrazumijeva posebne implementacijske karakteristike (npr. da li se aplikacija izvršava na serveru, osobnom računalu ili nekom drugom uređaju).
- **Autorizacijski server** – Server koji dodjeljuje pristupne tokene klijentu nakon uspješne provjere autentičnosti vlasnika resursa te dobivanje autorizacije.

Autorizacijski server može biti na istom serveru kao server resursa ili kao zaseban entitet. Osim toga, autorizacijski server može izdati pristupne tokene koji su prihvaćeni od strane nekoliko servera resursa.



Slika 7: Shema OAuth 2.0 toka protokola (Izvor: Web)

Klijent zahtijeva autorizaciju od vlasnika resursa. Zahtjev za autorizaciju se može izravno uputiti vlasniku resursa ili indirektno preko autorizacijskog servera kao posrednika. Klijent prima odobrenje za autorizaciju koja predstavlja autorizaciju vlasnika resursa korištenjem jednog od četiri načina odobrenja autorizacije. Vrsta odobrenja ovisi o načinu korištenja zahtjeva za autorizaciju te vrsti kojoj podržava autorizacijski server. Klijent zatražuje pristupni token autenticiranjem s autorizacijskim serverom te predočuje odobrenja za autorizaciju. Autorizacijski server autenticira klijenta i potvrđuje odobrenje za autorizaciju te ako je važeći onda izdaje pristupni token. Klijent zahtijeva zaštićeni resurs sa servera resursa te ga autenticira predočenjem pristupnog tokena. Server resursa potvrđuje pristupni token, te ako je valjan uslužuje zahtjev. Preporučena metoda za klijenta da dobije autorizacijsko odobrenje od vlasnika resursa je korištenjem autorizacijskog servera kao posrednika. (Hardt, 2012)

3.4.2 Autorizacijska odobrenja

Autorizacijsko dopuštenje su podaci koji predstavljaju ovlaštenje vlasnika resursa (za pristup svojim zaštićenim resursima) koje koristi klijent za dobivanje pristupnog tokena. Sastoji se od četiri tipova odobrenja – autorizacijski kôd, implicitni, lozinka vlasnika resursa i podaci klijenta .

3.4.2.1 Autorizacijski kôd

Autorizacijski kôd se dobiva pomoću autorizacijskog servera kao posrednik između klijenta i vlasnika resursa. Umjesto da se zahtjeva autorizacija izravno od vlasnika resursa, klijent usmjerava vlasnika resursa na autorizacijski poslužitelj gdje vlasnika resursa zauzvrat usmjerava klijentu sa autorizacijskim kodom. (Hardt, 2012)

Prije usmjeravanja vlasnika resursa prema klijentu sa autorizacijskim kôdom, autorizacijski server autentificira vlasnika resursa i dobiva autorizaciju. Budući da se vlasnika resursa autorizira samo preko autorizacijskog servera, podaci vlasnika resursa se nikada ne dijele sa klijentom. (Hardt, 2012)

Autorizacijski kôd pruža nekoliko važnih sigurnosnih prednosti kao što je sposobnost autentifikacije klijenta, prijenos pristupnog tokena izravno klijentu bez prolaska preko aplikacije vlasnika resursa gdje ga se potencijalno izlaže drugima, uključujući i vlasnika resursa. (Hardt, 2012)

3.4.2.2 Implicitni

Implicitno odobrenje je pojednostavljeni autorizacijski kôd optimiziran za klijente implementirane u pregledniku pomoću skriptnog jezika kao što je „JavaScript“. U implicitnom tijeku umjesto izdavanja autorizacijskog kôda klijentu, klijentu se izravno izdaje pristupni token (kao rezultat autorizacije vlasnika resursa). Vrsta odobrenja je implicitna budući da nema posredničkih podataka (kao što je to slučaj kod autorizacijskog kôda gdje se kasnije koristi za dobivanje pristupnog tokena). (Hardt, 2012)

Prilikom izdavanja pristupnog tokena tijekom implicitnog odobrenja, autorizacijski server ne autentificira klijenta. U nekim slučajevima, identitet klijenta se može provjeriti putem URI adrese preusmjeravanja koja se koristi za isporuku pristupnog tokena klijentu. Pristupni token može biti izložen vlasniku resursa ili drugim

aplikacijama sa kojima je pristup preko korisničkog posrednika vlasnika resursa. (Hardt, 2012)

Implicitna odobrenja poboljšavaju odaziv i učinkovitost nekih klijenta (npr. klijent koji je implementiran kao aplikacija u pregledniku) budući da smanjuje broj potrebnih odaziva za dobivanje pristupnog tokena. Međutim, ovu praktičnost treba izvagati sa sigurnosnim implikacijama korištenja implicitnog odobrenja pogotovo u slučaju kada je dostupan autorizacijski kôd kao vrsta odobrenja. (Hardt, 2012)

3.4.2.3 Lozinka vlasnika resursa

Lozinka vlasnika resursa (tj. korisničko ime i zaporka) može se koristiti izravno kao autorizacijsko odobrenje za dobivanje pristupnog tokena. Podaci se trebaju koristiti samo kada je visok stupanj povjerenja između vlasnika resursa i klijenta (npr. klijent je dio operativnog sustava uređaja ili vrlo povlaštena aplikacija) i kada druge vrste odobrenja nisu dostupne (primjerice autorizacijski kôd).

Iako ova vrsta odobrenja zahtijeva klijentu izravan pristup podataka vlasnika resursa, koriste se podaci vlasnika resursa za jedan zahtjev i zamjenjuju se za pristupni token. Ovaj tip odobrenja može eliminirati potrebu klijentu da pohrani podatke vlasnika resursa za buduću upotrebu, razmjenom podataka sa dugotrajnim pristupnim tokenom ili tokenom osvježavanja (*engl. Refesh token*). (Hardt, 2012)

3.4.2.4 Podaci klijenta

Podaci klijenta (ili drugi oblici autentikacije klijenta) se mogu koristiti kao autorizacijsko odobrenje kada je autorizacijska domena ograničena na zaštićene resurse pod kontrolom klijenta ili zaštićene resurse prethodno dogovorene sa autorizacijskim serverom. Podaci klijenta koriste se kao autorizacijsko odobrenje najčešće kada klijent djeluje u svoje ime (klijent je također vlasnik resursa) ili zahtijeva pristup zaštićenim resursima na temelju autorizacije prethodno dogovorene sa autorizacijski serverom. (Hardt, 2012)

3.4.3 Pristupni token

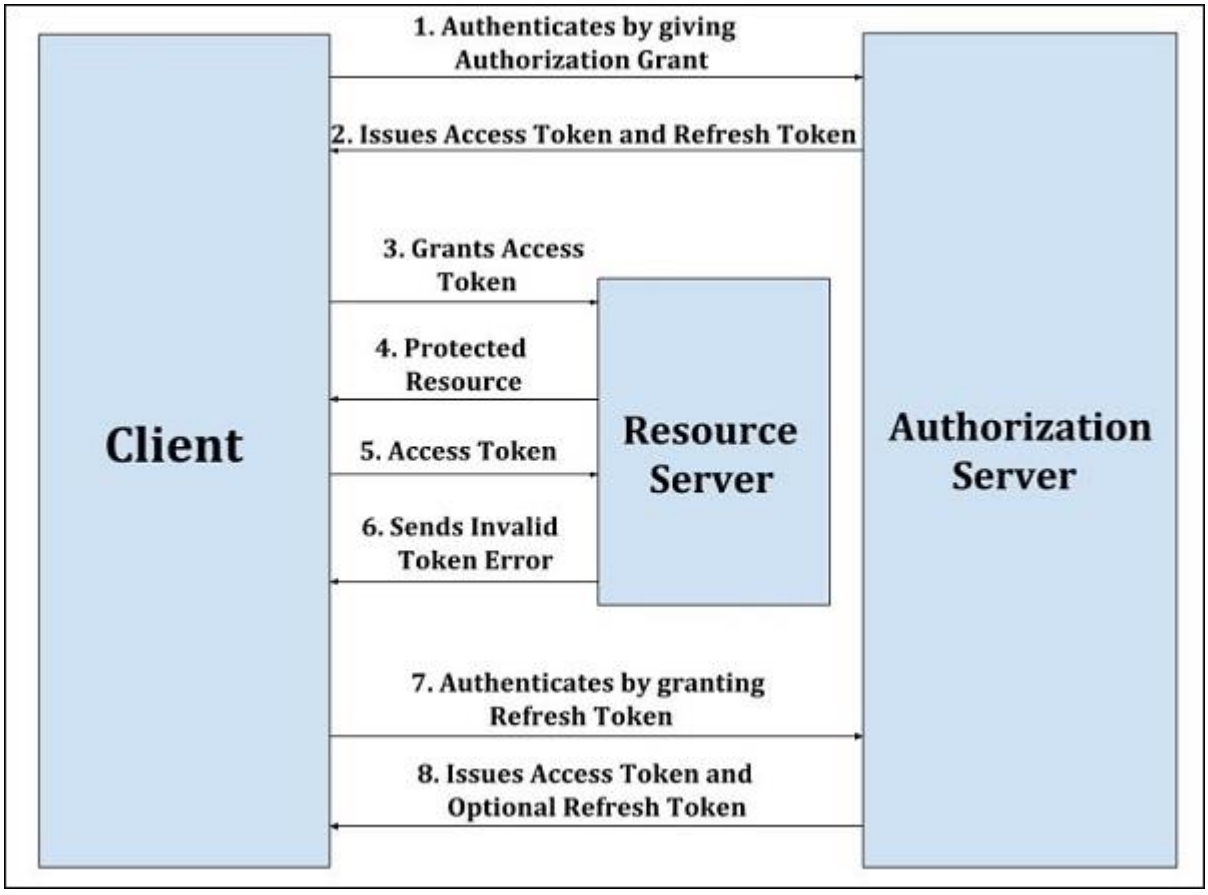
Pristupni tokeni su podaci koji se koriste za pristup zaštićenim resursima. To je znakovni niz koji predstavlja autorizaciju izdanu na klijent. Niz je obično vidljiv klijentu. Tokeni predstavljaju specifične domene i trajanja pristupa, koje dodjeljuje vlasnik resursa, te provodi server resursa i autorizacijski server. Token može označiti identifikator koji se koristi za preuzimanje autorizacije ili može sadržavati podatke o autorizaciji u provjerljivom obliku (tj. od niza znakova koji se sastoji od nekih podataka i potpis). Dodatni podaci za provjeru autentičnosti mogu biti potrebni da bi klijent mogao koristiti token. (Hardt, 2012)

Pristupni token pruža apstraktni sloj tako da zamjenjuje različite vrste autorizacije (npr. korisničko ime i lozinka) sa jednim tokenom razumljivom serveru resursa. Ta apstrakcija omogućuje izdavanje pristupnih tokena koji su restriktivniji od autorizacijskog odobrenja kao i uklanjanje potrebe razumijevanja širokog raspona metoda provjere autentičnosti za server resursa. Pristupni tokeni mogu imati različite formate, strukture i metode (npr. kriptografska svojstva) temeljene na sigurnosnim zahtjevima servera resursa. (Hardt, 2012)

3.4.4 Token osvježavanja

Tokeni osvježavanja su podaci koji se koriste za dobivanje pristupnog tokena. Oni se izdaju klijentu od strane autorizacijskog servera i koristi se za dobivanje novog pristupnog tokena kada trenutni pristupni token postaje nevažeći ili istekne, ili za dobivanje dodatnih pristupnih tokena sa identičnom ili užom domenom (pristupni tokeni mogu imati kraći vijek trajanja i manje prava od autoriziranog vlasnika resursa). Izdavanje tokena osvježavanja nije obavezno te je po izboru autorizacijskog servera. Ako autorizacijski server izda token osvježavanja on najčešće bude uključen pri izdavanju pristupnog tokena. (Hardt, 2012)

Token osvježavanja je znakovni niz koji predstavlja autorizaciju dodijeljenu klijentu od strane vlasnika resursa. Niz je obično vidljiv klijentu. Token označava identifikator koji se koristi za preuzimanje podataka o autorizaciji. Za razliku od pristupnog tokena, tokeni osvježavanja su namijenjeni samo za autorizacijski server i nikada se ne šalju na servere resursa. (Hardt, 2012)



Slika 8: Obnavljanje isteklog pristupnog tokena (Tutorials Point (I) Pvt. Ltd., 2018)

Obnavljanje isteklog pristupnog tokena se odrađuje preko tokena osvježavanja u sljedećim koracima: (Hardt, 2012)

1. Klijent zahtijeva pristupni token autenticiranjem sa autorizacijskim serverom ponuđujući autorizacijsko odobrenje.
2. Autorizacijski server autenticira klijenta i potvrđuje autorizacijsko odobrenje i ako je važeće izdaje pristupni token i token osvježavanja.
3. Klijent zahtijeva zaštićeni resurs predstavljanjem pristupnog tokena.
4. Server resursa potvrđuje pristupni token i ako je valjan uslužuje zahtjev.
5. Koraci 3. i 4. se ponavljaju dok ne istekne pristupni token. Ako je klijent saznao da je pristupni token istekao prelazi u 7. korak, inače odrađuje još jedan zahtjev za zaštićeni resurs.
6. Budući da je pristupni token nevažeći, server resursa vraća pogrešku nevažećeg tokena.

7. Klijent zahtijeva novi pristupni token autenticiranjem na autorizacijski server iznoseći token osvježavanja. Zahtjevi za provjeru autentičnosti klijenta temelje se na vrsti klijenta i na politici autorizacijskog servera.
8. Autorizacijski server autenticira klijenta i potvrđuje token osvježavanja a ako je valjan izdaje novi pristupni token, te po želji i novi token osvježavanja.

4 PLANIRANJE I PROJEKTIRANJE POZADINSKOG SERVISA

Za početnu verziju sustava koji bi rješavao problem redova čekanja tako da bi se smanjivalo potrebno vrijeme samog čekanja u redu nužno je imati pozadinski sustav koji bi implementirao logiku za smanjivanje vremena provedenog u redu čekanja. Ideja je iskoristiti najmodernije tehnologije ponuđene u tom trenutku koje bi poprilično uskratile vrijeme potrebno za izgradnju servisa. Izrada pozadinskog servisa koji bi bio neovisan i bez stanja omogućilo bi se na način da pozadinski servis bude po REST pravilima gdje određeni tip poziva bez stanja – kao što je HTTP protokol GET metoda, gađa krajnju točku servisa (*engl. endpoints*) te pokreće određenu akciju. To znači da bi takav REST servis imao *API* koji bi bio izložen na određenoj web adresi, što bi omogućavalo da se pristupa preko URL-a (*engl. Uniform Resource Locator*) sa nekog odvojenog sučelja web stranice ili da se taj poziv odradi sa drugog servisa te se pozove željena akcija na servisu za rješavanje problema redova čekanja.

Da bi servis bio cjelovit mora sadržavati web server, bazu podataka, metode autentikacije i autorizacije. Web server sa mnoštvom pomoćnih metoda za izgradnju produkcijskog servisa nudi „Spring Boot“ programski okvir, dok vrhunsko rješenje za metode autentikacije i autorizacije daje „Spring Security“ koji je često održavan od strane velike zajednice „Spring“ programskog okvira. Budući da je servis samo ideja te nije prethodno detaljno isplaniran, postoji idealna vrsta baze podataka gdje ne treba imati unaprijed striktno definiranu shemu modela podataka i njihovih veza. Takvu prednost nudi NoSQL baza podataka gdje ne treba definirati sheme niti strukture tablica već se podaci spremaju kao dokumenti u JSON formatu te se svaki podatak sprema ili dohvaća specijalnim naredbama dizajniranim za spremanjem i čitanjem velikih količina podataka u što kraćem vremenu. Na temelju podataka uspoređenih baza podataka prevagu su dale NoSQL baze podataka te između „CouchDB“ i „MongoDB“ prevagnuo je „MongoDB“ za bazu podataka budući da daje najbolje performanse te je najbolji izbor za brzo rastuće baze sa nedefiniranim setom podataka.

5 PRIMJENA ODABRANE TEHNOLOGIJE I IZRADA SERVISA

Izrada servisa za upravljanje redovima čekanja je odrađena u Java programskom jeziku te aplikaciji za integrirano razvojno okruženje (*IDE*) „IntelliJ IDEA 2017 Ultimate“. Korišten je web programski okvir „Spring Boot“ i „MongoDB“ kao NoSQL baza podataka. Servis za upravljanje redovima čekanja ima glavna načela REST servisa sa krajnjim točkama sa kojih klijent može odrađivati funkcije prijave i registracije, odabir ustanove, dodavanja u red na izabranom šalteru te otkazivanje uzetog broja.

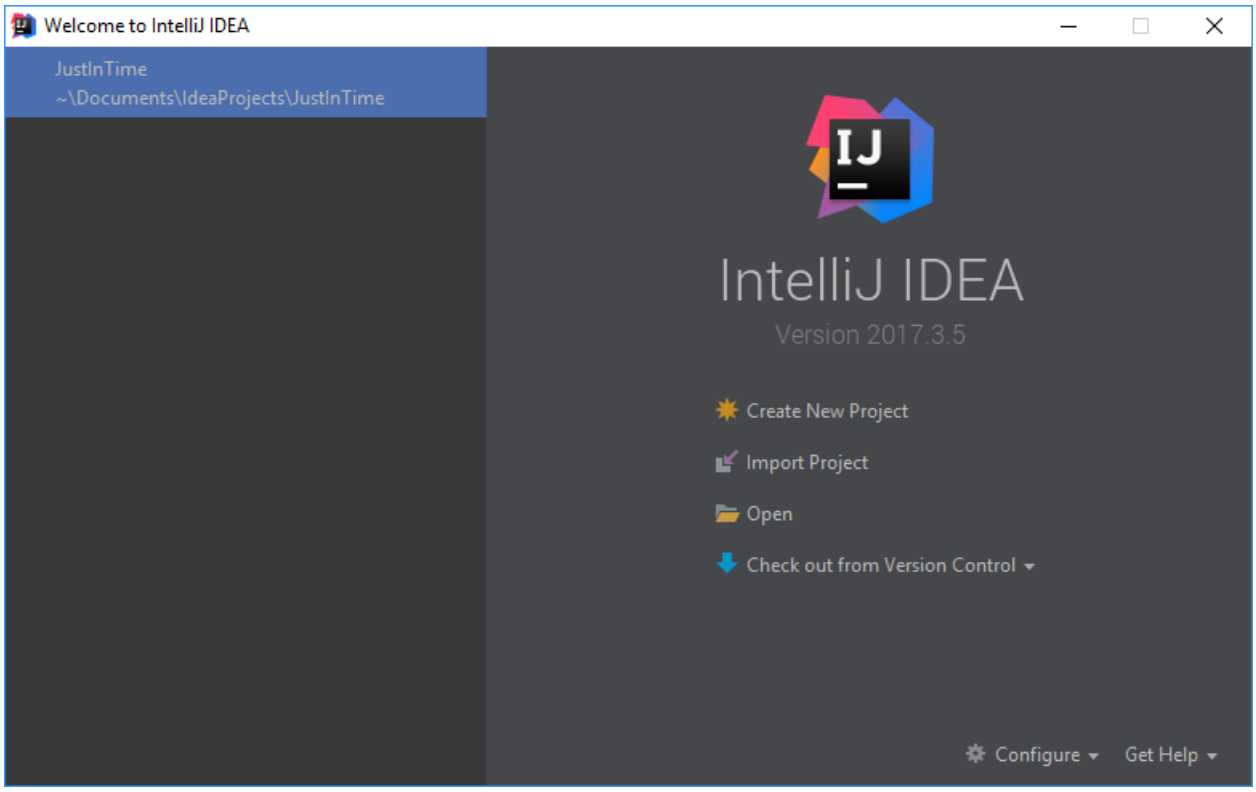
5.1 KONFIGURACIJA SPRING BOOT PROJEKTA

Prvi korak je bio istražiti postoji li određeni predložak koji se može iskoristiti kao kostur projekta. Postoji alat na stranici „Spring Boot“-a pod nazivom „Spring Initializr“ koji generira predložak prema zadanim postavkama. Od postavki se može birati koji programski jezik će se koristit gdje je na izboru Java, Kotlin, Groovy. Zatim se bira da li će biti „Maven“ ili „Gradle“ projekt te verzija „Spring Boot“-a koja će se koristiti kao i mnoštvo ostalih pomoćnih biblioteka koje će biti pridodane u generirani predložak projekta.

5.1.1 IntelliJ IDEA

Generirani projekt otvorimo u željenom *IDE*-u gdje će u ovom primjeru to biti „IntelliJ IDEA“ . Ona se pokazala kao jedan od najmoćnijih *IDE* aplikacija današnjice. Funkcija dovršavanja koda (engl. *Code completion*) je daleko ispred bilo kojih drugih *IDE*-a. Kontekstualno je svjesna i daje točne prijedloge 99% vremena. To je osobito korisno za vrlo rječite jezike kao što je Java. Nadalje, ima mogućnost analiziranja koda prilikom tipkanja. Pogreške su odmah naznačene, ali osim toga označava određeni dio koda koji je nedostupan ili može proizvesti neku pogrešku prilikom pokretanja. Također, ponuđuje savjete kako bi se mogla pojednostavniti logika i održavanje koda čistim. Refaktoriranje koda je veoma brzo, precizno i pouzdano. Jednostavan je za korištenje i sučelje je organizirano na vrlo intuitivan način. Izuzetno je stabilan, troši

malo resursa i daleko više povećava produktivnost naspram alternativnih *IDE*-a. (Quora, 2018)



Slika 9: Primjer početne stranice IntelliJ IDEA (autor, 2018.)

Nakon pokretanja iz generiranog projekta korisno je imati dodatke (*engl. plugins*) kako bi se olakšao i ubrzao rad na projektu. Budući da će se projekt oslanjati na „MongoDB“ bazu podataka za pohranu podataka potrebno je instalirati dodatak „Mongo Plugin“. Instalacija se izvršava preko „Settings“ opcije u „File“ izborniku, zatim pod menijem „Plugins“ klikne se „Browse repositories…“ i upiše željeni dodatak te se instalira. „Mongo Plugin“ omogućuje osnovno upravljanje „MongoDB“ bazama podataka. Osim toga, još jedan izuzetno koristan dodatak je „Lombok Plugin“. „Lombok“ je Java biblioteka koja se automatski spaja na *IDE* preko dodatka te nudi mnoštvo Java anotacija za automatsko generiranje koda prilikom pokretanja projekta. Uz „Lombok“ nije potrebno pisati *getter* i *setter* metode za modele već je dovoljno staviti anotaciju na sam model klase gdje će „Lombok“ sam generirati navedene metode. Najčešće korištene „Lombok“ anotacije: (Singh, 2017)

- @Getter – generira *getter* metode za čitanje vrijednost atributa
- @Setter – generira *setter* metode za postavljanje vrijednost atributa klase

- @NonNull – generira null provjere na označenim atributima klase
- @ToString – generira metodu *toString()* koja ispisuje sve atribute klase
- @EqualsAndHashCode – generira metode *equals* i *hashcode* sa dodatnim opcijama za proširenje ili isključenje za koje atribute klase će se odnositi
- @NoArgsConstructor – generira konstruktor bez parametra
- @RequiredArgsConstructor – generira konstruktor sa parametrima atributa klase koji su definirani kao *final* i nisu inicijalizirani te sa anotacijom @NonNull
- @AllArgsConstructor – generira konstruktor sa parametrima svih atributa klase
- @Data – generira g*etter*, *setter, equals, hashcode* i *toString* metode

5.2 MLAB I HEROKU

Ideja je da se servis za upravljanje redovima čekanja razvija lokalno na računalu te se koristi neki od *Git* servisa za verzioniranje izvora koda odnosno snimki verzija izvora koda. U ovom slučaju će to biti „GitHub“ gdje će se nalaziti kompletan izvorni kod projekta i njegove izračvane verzije preko „git branch“-eva. Budući da je u cilju imati servis koji bi bio uvijek dostupan te pristupiv sa bilo kojeg uređaja potrebno ga je postaviti na Internet. Taj problem se pojavljuje kod svakog IT start-upa ili tvrtke koje žele plasirati svoj proizvod ili softver na Internet. Neke tvrtke nađu rješenje u tome da si same postave potrebnu infrastrukturu za plasiranje i po potrebi skaliranje IT proizvoda, ali takvo je rješenje za start-up projekte ili male tvrtke najčešće preskupo te se okreću gotovim uslugama kao što su *IaaS*[12] ili *PaaS*[13]*.* Budući da neke od tih usluga imaju početne testne verzije koje nude besplatno za koristiti, odlično rješavaju problem da servis za upravljanje redovima čekanja bude pristupiv sa svih uređaja u bilo kojem vremenu.

[12] IaaS - Infrastructure as a Service se odnosi na hardversku infrastrukturu potrebnu za rad servisa

[13] PaaS – Platform as a Service se odnosi na proširive komponente u oblaku koji proširuju željeni servis

„Heroku“ je platforma kao servis (*engl. PaaS*) u oblaku temeljena na kontejnerima za postavljanje aplikacija ili servisa. Razvojni programeri koriste „Heroku“ za implementaciju, upravljanje i skaliranje modernih aplikacija. Platforma je fleksibilna i jednostavna za upotrebu, a programerima nudi najjednostavniji put za dobivanje aplikacija na tržište. „Heroku“ se može u potpunosti upravljati dajući programerima slobodu da se usredotoče na svoj razvojni proizvod bez ometanja održavanja poslužitelja, hardvera ili infrastrukture. (Heroku, 2018)

Na „Heroku“ će se nalaziti sam servis za upravljanje redovima čekanja odnosno biti će pokrenuti kao *REST* aplikacija te će se moći pristupiti preko *URL*-a. Također, baza podataka će biti postavljena na drugom servisu tako da sam servis može biti skalabilan te da ostale instance servisa imaju iste podatke kao i original. Za to će se koristiti „mLab“ koji nude *DaaS*[14] također besplatno za početnu i najosnovniju verziju usluge.

„mLab“ je servis za potpuno upravljanje baza podataka u oblaku sa naprednim mogućnostima kao što je automatizirano postavljanje i skaliranje baze podataka „MongoDB“, sigurnosno kopiranje i oporavak, sustav praćenja i upozorenja 24 sata na dan, alati za upravljanje webom i službu za stručnu podršku. „mLab“-ova platforma baze podataka kao servis nudi više stotina tisuća baza podataka preko „AWS[15]“-a, „Azure[16]“-a i „Google Compute Engine[17]“-a i omogućuje razvojnim programerima da se usredotoče na razvoj proizvoda umjesto na postavljanje arhitekture i potrebnih operacija. (mLab, 2018)

[14] DaaS – Database as a Service je mali dio IaaS-a gdje se mogu iznajmiti serveri za bazu podataka

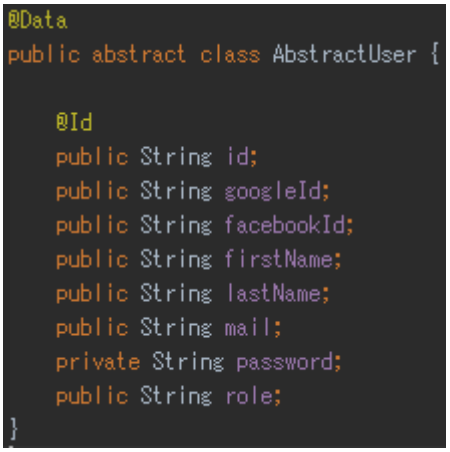
[15] Amazon Web Services je IaaS platforma u oblaku koja nudi računalnu snagu, pohranu podataka, isporuku sadržaja i druge funkcije potrebne u rastu i povećanju tvrtki. Datum prvog izdanja 2006.

[16] Microsoft-ova IaaS inačica AWS-a. Datum prvog izdanja 2010.

[17] Google-ova IaaS inačica AWS-a. Datum prvog izdanja 2011.

5.3 MODELI PODATAKA

Svaki servis koji koristi bazu podataka mora imati modele podataka koji će biti povezani sa modelima podataka u bazi podataka. Kako servis za upravljanje redova čekanja koristi „MongoDB“ bazu podataka, modeli će biti anotirani sa *@Document* što predstavlja Java anotaciju u „Spring Data“ biblioteci za rad sa entitetima u „MongoDB“ bazi podataka. Dodatno, svi entiteti će imati polje *id* koji će predstavljati vrijednost iz baze podataka te će biti anotirani sa *@Id*. Nadalje, svi modeli imaju definirane atribute u svojoj bazičnoj apstraktnoj klasi iz razloga što se nasljeđivanje dobiva na smanjenju redundancije koda te lakše održavanje budući da se promjene na bazičnoj klasi reflektiraju na svim ostalim klasama koje ih nasljeđuju.



Slika 10: Model korisnika (autor)

Svaki korisnik mora imati određene informacije koje ga opisuju da bi servis mogao raditi. Kako servis nudi više mogućnosti registracije ili prijave, potrebno je spremiti određene informacije vezane uz to – što su u ovom slučaju *id* za prijavu i registraciju iz samog servisa, *googleId* za prijavu ili registraciju preko „Google sign-in“ *API*-a te *facebookId* ako je korisnik to odradio preko „*Facebook sign-in*“ *API*-a. Osim toga, potrebne informacije o korisniku su ime i prezime, email adresa, zaporka koja je spremljena preko sigurnosne *hash* funkcije te njegova uloga koja će se koristiti za prava na određenim resursima, odnosno *API* točkama.

```java
@Data
public abstract class AbstractFacility {
    @Id
    protected String id;
    protected String name;
    protected String address;
    protected String mail;
    protected String telephone;
}
```

Slika 11: Apstraktni model ustanove (autor)

Potrebne informacije za ustanove su definirane u apstraktnoj klasi te se sastoje od naziva, adrese, emaila i telefonskog broja.

```java
@Document
public class Facility extends AbstractFacility {
    public ArrayList<Queue> queues = new ArrayList<>();
}
```

Slika 12: Model ustanove (autor)

Ustanove nasljeđuju njihov bazični model te imaju atribute isto kao i bazični model sa dodatnim atributom za redove koji će u ovom slučaju služiti kao šifrarnik za popis svih redova određene ustanove.

```java
@Data
public abstract class AbstractQueue {
    @Id
    public String id;
    public String name;
}
```

Slika 13: Apstraktni model reda (autor)

Apstraktna klasa reda drži osnovnu informaciju o redu, a to je naziv reda.

```java
@Data
@Document
public class QueuePriority extends AbstractQueue {
    private Instant openingTime;

    private Instant closingTime;

    public int priority = 0;
}
```

Slika 14: Model reda (autor)

Model reda koji nasljeđuje bazičnu klasu reda te time dobiva i naziv reda ima definirane atribute za vremena otvaranja i zatvaranja reda koji se postavljaju prilikom

dolaska i odlaska šalterskog službenika te prioritet koji predstavlja redni broj u određenom redu.

5.4 OAUTH2 STANDARD ZAŠTITE

Pristup krajnjih točki REST servisa za upravljanje redovima čekanja je bez dodatnog konfiguriranja otvoren za javnost budući da svatko može pristupiti *API*-ju te izvršiti akcije. Takvo ponašanje najčešće nije korisno osim u nekim ranim fazama razvoja sustava. Budući da će servis za upravljanje redovima čekanja nuditi određene osjetljive akcije gdje bi trebalo ograničiti tko može izvršiti takvu akciju, potrebno je zaštiti REST API određenim sustavom za sigurnost. Budući da se servis oslanja na „Spring“ programske okvire koristit će se „Spring Security“ za postavljanje sigurnosti na REST API-ju. Koristit će se OAuth2 standard zaštite budući da on pruža najviše standarde sigurnosti današnjice. Ideja je da se korisnik mora autenticirati na određenoj krajnjoj točki servisa gdje će nakon implementacije OAuth2 to biti autorizacijski server. Nakon autenticiranja sa autorizacijskim serverom korisnik će dobiti pristupni token koji će biti na svakom zahtjevu prema serveru resursa, na kojemu se nalaze krajnje točke servisa za upravljanje redovima čekanja.

5.4.1 Konfiguriranje autorizacijskog servera

Autorizacijski server vrši autentikaciju korisnika te uspješno autenticiranim i autoriziranim korisnicima vraća pristupni token koji predstavlja dozvolu za pristup resursima na serveru resursa. Pristupni token se dobije sa ove krajnje točke servisa: */oauth/token?grant_type=password&username=tperkovic@unipu.hr&password=******

Implementira se na način da se napravi konfiguracijska klasa sa *@Configuration* koja će naslijediti *AuthorizationServerConfigurerAdapter* te će implementirati *configure* metode za *ClientDetailsServiceConfigurer*, *AuthorizationServerEndpointsConfigurer i AuthorizationServerSecurityConfigurer.* Klasi će biti potreban A*uthenticationManagerBean* koji će se ubrizgati sa *@Autowire* anotacijom da bi klasa mogla koristiti *bean.* Na napravljenoj klasi potrebno je staviti

anotaciju *@EnableAuthorizationServer* koja omogućuje da se pokrene autorizacijski server prilikom pokretanja aplikacije.

Implementacija metode *configure* sa parametrom *ClientDetailsServiceConfigurer* je napravljena tako što se pozivaju metode klase *ClientDetailsServiceConfigurer* za konfiguraciju klijenta u memoriji sa definiranim nazivom „trusted-client“, tipom autorizacijskog odobrenja „password“, „authorization_code“, „refresh_token“, „implicit“, zatim postavljanja *authorities* na „ROLE_CLIENT“ i „ROLE_TRUSTED_CLIENT“, postavljanje *scope* na „trust“, definiranjem *secret* sa „secret“ te postavljanju trajanju pristupnog tokena na jedan sat i tokena osvježenja na dvanaest sati.

Potrebno je definirati *bean token store* koji služi za čuvanje kreiranih pristupnih tokena i nudi određene akcije nad tokenima. U slučaju servisa za upravljanje redovima čekanja će se koristiti *JwtTokenStore* koji u biti ne perzistira kreirane tokene već nudi metode za čitanje tokena i konverzije tokena u i iz autentikacije. Osim toga, treba definirati *bean* koji implementira logiku konverzije tokena iz autentikacije i u autentikaciju pomoću simetričnoga ključa.

Primarni *bean DefaultTokenServices* je obavezan budući da je to glavni dio koji će kreirati pristupne tokene i tokene osvježenja. Također preko njega se može definirati da li želimo da nam se generira token osvježenja koji je opcionalan, te se mora postaviti *TokenStore* koji će definirati da li će se tokeni perzistirati te način konverzije tokena u autentikaciju i obrnuto.

Implementacija metode *configure* sa parametrom *AuthorizationServer EndpointsConfigurer* definira ponašanje krajnjih točaka servisa te se ovom metodom postavlja da krajnje točke servisa koriste *tokenStore* koji je prethodno definiran, definiraju kojim metodama zahtjeva će se moći pristupiti zahtjevu za pristupni token, postavlja se prethodno definiranu konverziju *TokenStore-a* za konverziju pristupnih tokena te se prosljeđuje AuthenticationManagerBean koji upravlja sa zahtjevima tako da se omotaju metodama za autentikaciju.

Implementacija metode *configure* sa parametrom *AuthorizationServerSecurity Configurer* definira se ponašanje krajnje točke za zahtijevanje pristupnog tokena u kojoj se dozvoli ili zabrani pristup svima, ili pojedinoj grupi te koju metodu će pozvati za autentikaciju.

5.4.2 Konfiguracija adaptera za web sigurnost

Kada klijentska aplikacija da zahtjev za pristupni token na autorizacijski server ova konfiguracijska klasa će definirati način autentikacije za pristupni token. U ovom slučaju će to biti autentikacija putem korisničkog imena i lozinke.

Konfiguracija se vrši na način da se kreirana klasa anotira sa *@Configuration* i time označi „Spring Boot“-u da će to biti konfiguracijska klasa, te se onda naslijedi klasa *WebSecurityConfigurerAdapter*. Klasa se sastoji od novo kreiranih *bean*-ova *inMemoryUserDetailsManager* i *passwordEncoder*. Klasa ubrizgava *bean*-ove *userRepository* koji će se koristiti za dohvaćanje korisnika iz baze podataka te učitavanju u memorijski menadžer korisničkih detalja, *passwordEncoder* koji će postaviti vrstu enkripcije za lozinke*, globalUserDetails* u kojem će autentikacijski menadžer postaviti vrstu enkripcije lozinke dodajući *bean*-ove pomoću *@Autowired* anotacije iz „Spring Boot“ aplikacijskog konteksta.

Glavni *bean* u ovoj klasi je *inMemoryUserDetailsManager* u kojemu se učita pomoću korisničkog repozitorija cijela baza podataka u obliku liste korisnika te se dodaje u obliku korisničkog imena, lozinke i uloge u objekt I*nMemoryUserDetailsManager.* To će omogućiti da prilikom pokretanja servisa za upravljanje redovima čekanja budu učitani svi korisnici iz baze podataka te će se moći prijavljivati sa svojim korisničkim imenom i lozinkom.

Bean passwordEncoder će implementirati objekt *BCryptPasswordEncoder* što će omogućiti da kodiranje lozinke bude vrste „BCrypt“.

Ubrizganom *bean-u globalUserDetails* se metodom postavljaju korisnički detalji na kreirani *bean inMemoryUserDetailsManager* koji drži učitane korisnike iz baze podataka i novo definirani *passwordEncoder* koji omogućuje „BCrypt“ kodiranje.

5.4.3 Konfiguracija servera resursa

Server resursa ima iza sebe krajnje točke servisa koje korisnik želi pozvati. On može dodatno konfigurirati određene akcije prije samog poziva resursa. Među ostalim, server resursa u konačnici se ne mora nalaziti na istoj fizičkoj mašini kao što je autorizacijski server. Ako su na različitim mašinama potrebno je postaviti *TokenStore* na *RemoteTokenStore* te konfigurirati na autorizacijskom serveru i serveru resursa da čitaju isti *TokenStore.* U slučaju ovoga servisa za upravljanje redovima čekanja server resursa i autorizacijski server se nalaze na istoj mašini.

Konfiguracija servera resursa se vrši na način da se kreira konfiguracijska klasa sa anotacijom *@Configuration* te naslijedi *ResourceServerConfigurerAdapter*. Potrebno je implementirati metode *configure* sa parametrom *ResourceServerSecurity Configurer* te istu tu metodu sa parametrom *HttpSecurity*. Osim toga, potrebno je definirati iste *bean*-ove kao i kod autorizacijskog servera a to su *TokenStore, JwtAccessTokenConverter i DefaultTokenServices.*

Bean JwtAccessTokenConverter implementira konverziju pristupnog tokena iz i u autentikaciju sa definiranim simetričnim ključem. Ključ i konverzija mora biti ista kao i kod autorizacijskog servera tako da autorizacijski server i server resursa mogu pretvarati token u autentikaciju i obrnuto na isti način.

Bean TokenStore implementira isti *JwtTokenStore* sa proslijeđenim *bean*-om *JwtAccessTokenConverter* kao i kod autorizacijskog servera.

Primarni *Bean DefaultTokenServices* kreira objekt na koji se postavlja kreirani *bean TokenStore.* Svrha mu je da razumije dobivene pristupne tokene od autorizacijskog servera na način da ih može konvertirati u autentikaciju i pročitati.

U metodi *configure* sa parametrom *ResourceServerSecurityConfigurer* se postavlja *tokenService* na kreirani primarni *bean DefaultTokenServices* da bi server resursa mogao očitati postavke.

U metodi *configure* sa parametrom *HttpSecurity* definira se ponašanje http zahtjeva prema krajnjoj točki resursa. Postavlja se vrsta sesije na *STATELESS* preko *sessionManagement* metode, preko metode *antMatchers* definira se lista *URI*-a koji imaju otvoreni pristup te se za preostale *URI*-je koji nisu na listi vrši autorizacija i

autentikacija zahtjeva. Time je definirano koji resursi su dostupni svima a za koje je potrebno biti prijavljen u servis za upravljanje redovima čekanja.

5.4.4 Postavljanje metode globalne sigurnosti

Definiranjem ove klase moći će se specifičnije definirati ograničenja na krajnjim točkama servisa budući da će se moći koristiti varijable u anotacijama *@PreAuthorize, @PostAuthorize* na kojima će se definirati ograničenje po određenom atributu korisnika. Servis za upravljanje redovima čekanja će koristiti ulogu kao skup dozvoljenih prava na određene krajnje točke servisa. Potrebno je anotirati samo one krajnje točke servisa koje nisu javne te je potrebna prijava u sustav.

Potrebno je definirati konfiguracijsku klasu sa anotacijom *@Configuration* te anotaciju @EnableGlobalMethodSecurity koja će omogućiti da se na pokretanju servisa učita i koristi ova konfiguracijska klasa za server resursa. Osim toga, konfiguracijska klasa mora naslijediti klasu *GlobalMethodSecurityConfiguration* gdje će se preklopiti metoda i implementirati *createExpressionHandler*. Potrebno je samo napraviti da se kreira objekt klase *OAuth2MethodSecurityExpressionHandler* koja ima u sebi implementirano čitanje varijabli iz anotacija *@PreAuthorize, @PostAuthorize* i sve će raditi po definiciji.

5.5 REGISTRACIJA KORISNIKA

Da bi se omogućilo korisniku da istovremeno može uzeti broj za red u više ustanova i redova u ustanovi i iste te brojeve provjeravati na različitim uređajima, te možda i sačuvati listu najčešće posjećenih ustanova nužno je koristiti pristup registracije i prijavljivanja korisnika u sustav.

Servis nudi tri opcije registracije korisnika:

1. Registracija preko servisa
2. Registracija preko Google računa
3. Registracija preko Facebook računa

5.5.1 Registracija preko servisa

Registracija korisnika odrađuje se preko REST API poziva */user/create.* Korisnik upisuje osnovne podatke kao što su ime i prezime, email adresu te lozinku. Prije nego se korisnik sačuva u bazu podataka „MongoDB*“*, mora se provjeriti da li već postoji tako da se ne bi isti korisnik više puta registrirao. To je odrađeno na način da se provjeri korisnikova email adresa u bazi podataka, što ujedno znači da je svaki korisnik vezan za jedinstvenu email adresu. Ako je korisnik već registriran s tim emailom, onda servis vraća „Spring Boot“ odgovor tipa „ResponseEntity“ sa http statusom *CONFLICT* (kod 409).

Nakon što je korisnik upisao sve podatke, lozinka se kriptira „BCrypt“ raspršnom (*engl. hash*) funkcijom koja je temeljena i izvedena na „Blowfish“ blok šifri. „BCrypt“ je korišten iz razloga što nema za sad poznate slabosti, a vremenska kompleksnost za napad uzastopnim pokušavanjem (*engl. Brute force*) je dovoljno velik da se ne isplati raditi, pa čak uz pomoć skupine računala u oblaku koja mogu zajedničkim snagama izračunavati permutacije lozinka.

Svaki novo registrirani korisnik ima početnu postavljenu vrijednost uloge (*engl. Role*) na „USER“ sa ograničenim pravima što omogućuje razinu sigurnosti da se izbjegne slučajno ili zlonamjerno iskorištavanje prava izvan predviđenih.

Kreiranje korisnika je odrađeno pomoću klase *InMemoryUserDetailsManager* iz paketa „Spring Security“ što je u stvari implementacija *UserDetailsManager* sučelja koja nudi osnovne operacije za registraciju korisnika. Ta implementacija sučelja omogućuje kreiranje korisnika unutar memorije u obliku mape sa metodama za kreiranje i ažuriranje korisnika, promjenu lozinke te brisanje korisnika.

Nakon uspješnog kreiranja računa korisnika vraća se odgovor klijentu sa podacima koje je unio te kriptiranom lozinkom i http status *CREATED*.

5.5.2 Registracija preko Google računa

Krajnja točka REST API-a se nalazi na */user/google* gdje se ujedno novi korisnik registrira na sustav za upravljanje redovima čekanja automatski ili, ako je već bio registriran od prije, prijavljuje u sustav. Za registraciju ili prijavljivanje u sustav potrebno je imati Google-ov token koji se dobije prijavljivanjem u Google-ov sustav preko definiranog API-ja. Nakon uspješnog prijavljivanja te dobivanja valjanog tokena, token se prosljeđuje na REST API sustava za upravljanje redovima čekanja gdje se iz njega pomoću Google-ove biblioteke poziva metoda za provjeru tokena *GoogleIdTokenVerifier.verify()*. Ako je token valjan, iz njega se može pomoću metode *GoogleIdToken.getPayload()* dobiti javne informacije o korisniku ovisno o tome što je korisnik postavio na Google računu. Ako token nije valjan, servis vraća *ResponseEntity* sa http statusom *NOT_FOUND*.

Servis za identifikaciju korisnika koristi *googleId,* budući da email nije striktno vezan za Google-ov račun korisnika. Tako da se prvo pretražuje da li postoji već korisnik sa tim *googleId*-om u bazi korisnika preko *UserRepository*-a te, ako ga nema, dodatno se još pretražuje po emailu. Ako takvog korisnika nema ni u tom trenutku, onda se pretpostavlja da je novi korisnik te se izvršava njegova registracija sa podacima dobivenih iz metode *getPayload().* U slučaju da je pronađen korisnik u bazi podataka sa proslijeđenim *googleId*-em ili emailom, kreira se pristupni token preko autorizacijskog servera nudeći mu kreirano autorizacijsko odobrenje. Nakon uspješnog kreiranja pristupnog tokena servis vraća odgovor *ResponseEntity* sa pristupnim tokenom i http statusom *OK.*

5.5.3 Registracija preko Facebook računa

Krajnja točka REST API-a se nalazi na */user/facebook* gdje ima sličnu logiku kao i sa Google računom. Znači ako korisnik prvi put pristupa servisu, onda se registrira automatski sa dobivenim podacima iz tokena, a ako ne onda se prijavljuje na sustav. Za registraciju ili prijavu na servis potrebno je imati *Facebook-ov* token koji se prosljeđuje na servis za upravljanje redovima čekanja, a dobiva ga se preko *Facebook API*-a prijavom na *Facebook*-ov servis. Servis za upravljanje redovima čekanja koristi *Facebook*-ovu biblioteku za dobivanje informacija o korisniku putem klase *FacebookTemplate,* u kojoj prilikom instanciranja objekta putem konstuktora klase proslijedimo token generiran od strane „Facebook Sign in“ API-a.

FacebookTemplate klasa osim metoda za rad sa Facebook računom nasljeđuje i implementira *AbstractOAuth2ApiBinding* koja ima implementaciju metode provjere tokena te metode za povezivanjem sa REST servisom koji su u ovom slučaju Facebook servisi.

Prije poziva metode za dobivanje informacija o korisniku potrebno je definirati koje informacije su nam potrebne. Za servis za upravljanje redovima čekanja dovoljne su informacije ime i prezime, email adresa te da li je verificirani korisnik. Pozivom metode *Facebook.fetchObject(„me“)* sa definiranim poljem informacija koje su potrebne za servis dobiva se trenutni prijavljeni *FacebookUser*. Servis ima provjeru da li su uspješno dobivene informacije o Facebook korisniku te ako nisu se vraća *ResponseEntity* sa porukom „*Facebook user info not received!*“ te http statusom *NOT_FOUND.* Dodatno se još provjerava da li je korisnik verificiran te ako nije servis vraća *ResponseEntity* sa porukom „*Facebook user is not trusted!*“ i http status *FORBIDDEN.*

Slično kao i kod registracije putem Google računa, prvo se provjerava da li postoji u bazi podataka poseban identifikator specifičan i unikatan za svaki Facebook račun, te ako ne postoji se dodatno provjerava da li postoji email u bazi podataka. Tek u slučaju da nema u bazi podataka takvog identifikatora i emaila onda se pretpostavlja da je korisnik novi, te se registracija odrađuje automatski i popunjavaju podaci dobiveni iz tokena preko *fetchObject()* metode i zatraženih informacija. Informacije o korisniku koje su izložene *Facebook API*-u definira sam korisnik tako što se prijavom na svoj Facebook račun podesi koja informacija je javna a koja nije. Ako je neka informacija o korisniku privatna *Facebook API* će vratiti *null* umjesto informacije. U slučaju da u bazi podataka već postoji korisnik sa *facebookId*-em onda se kreira autorizacijsko odobrenje specijalno za takvog korisnika te se prosljeđuje autorizacijskom serveru gdje on vraća valjani pristupni token za servis upravljanja redovima čekanja koji se na dalje koristi na svakom zahtjevu korisnika.

5.6 AKCIJE KORISNIKA

Servis za upravljanje redovima čekanja nudi akcije za korisnike koji su registrirani ili prijavljeni u sustav putem *REST* kontrolera. Za svaku akciju potrebno je postaviti metodu zahtjeva (*engl. RequestMethod*) na definiranu vrijednost da bi zahtjev mogao biti obrađen od strane servisa. Za neke akcije potrebno je imati ulogu *ADMIN* da bi se akcija mogla koristiti. Servis za upravljanje redovima čekanja ima akcije izložene na sljedećim krajnjim točkama:

- *getCurrentUser*: */user/me* sa metodom zahtjeva *GET*
- *getAllUsers*: */user/read-all* sa metodom zahtjeva *GET*
- *getUserByMail*: */user/{mail}* sa metodom zahtjeva *GET*
- *updateUser: /user/update/{id}* sa metodom zahtjeva *PUT*
- *deleteUser: /user/delete/{id}* sa metodom zahtjeva *DELETE*

5.6.1 Dohvaćanje trenutačnog korisnika

Ova akcija sa nazivom *getCurrentUser* se nalazi na krajnjoj točki servisa */user/me* te se pristupa sa *GET* metodom zahtjeva. Ono što se zaista dogodi je vrlo jednostavno. Kada servis dobije zahtjev učita se iz sigurnosnog konteksta *OAuth2* trenutačnog prijavljenog korisnika tako što se u metodi kao argument preda *Principal* klasa gdje kontroler zna sam automatski popuniti iz sigurnosnog konteksta. Metodom *Principal.getName()* se dobije korisničko ime trenutačno prijavljenog korisnika gdje se zatim pretraži preko korisničkog repozitorija u bazi podataka mail korisnika te se vrati objekt *User* sa popunjenim informacijama. Servis vrati *ResponseEntity* sa pronađenim *User* objektom i http status *OK*, a ako slučajno repozitorij nije pronašao korisnika sa takvim mailom (u međuvremenu obrisan) vrati *ResponseEntity* sa http statusom *NOT_FOUND*.

5.6.2 Dohvaćanje svih korisnika

Akcija se naziva *getAllUsers* i nalazi se na krajnjoj točki servisa */user/read-all* te se pristupa sa *GET* metodom zahtjeva. Za pristup toj akciji potrebno je imati ulogu *ADMIN*. Ova akcija je u biti pregled u bazu korisnika gdje se dobije lista svih *User* objekta sa njihovim informacijama. Ako je slučajno ta lista prazna, servis za upravljanje redovima čekanja vrati kao odgovor *ResponseEntity* sa http statusom *NOT_FOUND*, inače vrati listu tih objekata i http status *OK.*

5.6.3 Dohvaćanje korisnika po emailu

Akcija se naziva *getUserByMail* i nalazi se na krajnjoj točki servisa */user/{mail}* te se pristupa sa *GET* metodom zahtjeva. Mail u vitičastim zagradama označava da se uvijek mora predati kao parametar mail da bi kontroler prihvatio zahtjev jer inače ga odbije sa opisnom iznimkom gdje se može vidjeti zašto nije zahtjev prošao. I za ovu akciju je potrebno imati *ADMIN* ulogu da bi se akcija mogla izvršiti. Akcija je u biti metoda na korisničkom repozitoriju gdje se pretražuje baza podataka sa danim mailom i ako pronađe vrati objekt *User* sa informacijama o korisniku. Nadalje, kontroler vrati odgovor *ResponseEntity* sa tim popunjenim *User* objektom i http statusom *OK*, a u slučaju da nije pronašao korisnika sa tim mailom onda se vrati http status *NOT_FOUND*.

5.6.4 Ažuriranje korisnika

Akcija ažuriranja korisnika *updateUser* se nalazi na krajnjoj točki servisa */user/update/{id}* te se pristupa sa *PUT* metodom zahtjeva. Ova akcija ima obavezan parametar *id* što predstavlja identifikator u bazi podataka. Ostali podaci koji se prosljeđuju u obliku *User* objekta su opcionalni. Ideja je da se prosljeđuju samo oni atributi klase *User* koje bi korisnik htio promijeniti. Na primjer, ako korisnik želi promijeniti prezime treba se napraviti *PUT* zahtjev na krajnju točku servisa koji bi izgledao ovako */user/update/12001?lastName=Perković*. Prvo što servis odradi je da provjeri postoji li korisnik sa proslijeđenim *id*-em u bazi podataka. Ako ga ne pronađe onda se vrati odgovor *ResponseEntity* sa http statusom *NOT_FOUND*. Ako postoji, onda se mijenjaju atributi tog *User* objekta na one koji su proslijeđeni u parametru. Ako je proslijeđena korisnička lozinka onda će se prije promjene kriptirati „BCrypt“ jednokratnom *hash* funkcijom te će se kao takva spremati u bazu podataka. Ako je pronađeni korisnik od prije imao postavljenu ulogu bit će samo kopirana te se uloga ne može promijeniti preko *updateUser* akcije zbog sigurnosnih razloga. Osim što se ažurirani korisnik mijenja u bazi podataka, također je potrebno ažurirati tog korisnika koji je trenutno učitan u memoriju od samog startanja servisa. Kao što se na kreiranju korisnika pozivala metoda *createUser* u klasi *InMemoryUserDetailsManager* tako će se za ažuriranje iz te klase pozivati *updateUser* koji će ažurirati trenutnog korisnika u mapi učitanih korisnika. Nakon što je servis sve uspješno odradio vraća odgovor *ResponseEntity* sa ažuriranim objektom *User* i http statusom *OK*.

5.6.5 Brisanje korisnika

Akcija brisanja korisnika *deleteUser* se nalazi na krajnjoj točki servisa */user/delete/{id}* te se pristupa sa *DELETE* metodom zahtjeva. Akcija ima obavezan parametar *id* što predstavlja identifikator u bazi podataka. Za pristup ovoj akciji potrebno je biti prijavljen sa *ADMIN* ulogom. Akcija je u biti vrlo jednostavna jer je ideja samo brisanje postojećega korisnika iz baze podataka. Servis za upravljanje redovima čekanja preko korisničkoga kontrolera prosljeđuje *id* te se provjerava preko korisničkoga repozitorija da li je taj *id* postoji. Ako ne postoji servis vraća odgovor *ResponseEntity* sa http statusom *NOT_FOUND*, inače izvrši *delete* metodu u korisničkom repozitoriju te vrati izbrisani *User* objekt kao odgovor i http status *OK.*

5.7 AKCIJE USTANOVA

Akcije ustanova sadrže javni *API* za koji ne treba biti registriran, a osim toga ima i određene akcije za koje treba biti registriran sa ulogom *ADMIN* da bi im se moglo pristupiti. Za pristup javnom *API*-u ustanova ne treba biti prijavljen u sustav za upravljanje redovima čekanja iz razloga što informacije o ustanovi nisu sigurnosno osjetljive niti privatne, već sve te informacije su dostupne javnosti. Javni dio akcija se nalazi iza sljedećih krajnjih točaka servisa:

- *getFacility*: */facility/{id}* sa metodom zahtjeva *GET*
- *getAllFacilities*: */facility/read-all* sa metodom zahtjeva *GET*

Za pristup sljedećim krajnjim točkama servisa potrebno je biti registriran i prijavljen u servis za upravljanje redovima čekanja sa *ADMIN* ulogom da bi se akcije mogle uspješno izvršiti:

- *createFacility: /facility/create* sa metodom zahtjeva *POST*
- *updateFacility: /facility/update/{id}* sa metodom zahtjeva *PUT*
- *deleteFacility: /facility/delete/{id}* sa metodom zahtjeva *DELETE*

5.7.1 Dohvaćanje ustanove

Akcija dohvaćanja ustanove *getFacility* se nalazi na krajnjoj točki servisa */facility/{id}* te se pristupa sa *GET* metodom zahtjeva. Da bi servis vratio odgovarajući podatak, kontroler ustanova očekuje identifikator iz baze podataka u zahtjevu. *Id* iz zahtjeva se prosljeđuje na repozitorij ustanova gdje se zatim pretraži baza podataka ustanova sa proslijeđenim *id*-em. Ako repozitorij pronađe podatak, onda ga vrati te proslijedi kontroleru gdje se zatim kreira odgovor *ResponseEntity* sa pronađenim podatkom i http statusom *OK*, a ako ga ne pronađe onda se vrati odgovor sa http statusom *NOT_FOUND*.

5.7.2 Dohvaćanje svih ustanova

Akcija dohvaćanje ustanova *getAllFacilities* se nalazi na krajnjoj točki servisa */facility/read-all* te se pristupa sa *GET* metodom zahtjeva. Ova akcija na zahtjevu preko repozitorija ustanova vrati listu svih ustanova koje su u bazi podataka, a u slučaju da je baza podataka prazna vrati se prazna lista. Lista ustanova se proslijedi na *ResponseEntity* sa http statusom *OK,* a ako je prazna onda kontroler vrati odgovor sa http statusom *NOT_FOUND.*

5.7.3 Kreiranje ustanove

Akcija kreiranja ustanove *createFacility* se nalazi na krajnjoj točki servisa */facility/create* te se pristupa sa *POST* metodom zahtjeva. Akciji mogu pristupiti samo prijavljeni korisnici s *ADMIN* ulogom. Akcija na kontroleru ustanova očekuje parametre s nazivima atributa klase *Facility* koji se prilikom prosljeđivanja mapiraju na *Facility* objekt. Proslijeđeni parametri su opcionalni što znači da se na primjer može dodati ustanova samo sa parametrom *name* gdje bi zahtjev na servis izgledao ovako: */facility/create?name=MUP*. Nakon prosljeđivanja željenih parametara za *Facility* objekt, on se preko repozitorija ustanova sačuva u bazu podataka ustanova te kontroler vrati odgovor *ResponseEntity* sa kreiranim objektom ustanova te http statusom *CREATED*.

5.7.4 Ažuriranje ustanove

Akcija ažuriranja ustanove *updateFacility* se nalazi na krajnjoj točki servisa */facility/update/{id}* te se pristupa sa *PUT* metodom zahtjeva. Kontroler ustanova za ovu akciju očekuje obavezan parametar *id* koji je u biti identifikator ustanove u bazi podataka ustanova. Kao i kod akcija kreiranja ustanove, tako i kod akcije ažuriranja ustanove potrebno je imati *ADMIN* ulogu za pristup ovoj krajnjoj točki servisa. Sa akcijom ažuriranje ustanove mogu se mijenjati postojeći podaci uneseni za određenu ustanovu ili nadodavati podatke koji prethodno nisu uneseni kod akcije kreiranja ustanove. Prije izmjene ili nadopunjavanja podataka servis će proslijeđeni identifikator provjeriti preko repozitorija ustanova da li takav zapis postoji u bazi podataka ustanova. Ako ne postoji onda servis vraća odgovor *ResponseEntity* sa http statusom *NOT_FOUND*, inače će se izmijenjeni ili nadopunjeni podaci aplicirati na pronađeni objekt *Facility* te takav ažurirati preko repozitorija ustanova u bazi podataka ustanova.

5.7.5 Brisanje ustanove

Akcija brisanje ustanove *deleteFacility* se nalazi na krajnjoj točki servisa */facility/delete/{id}* te se pristupa sa *DELETE* metodom zahtjeva. Također i za pristup ove akcije potrebno je imati *ADMIN* ulogu. Akcija očekuje obavezan parametar *id* koji je identifikator *Facility* objekta u bazi podataka ustanova. Servis upravljanja redovima čekanja će proslijeđeni *id* pretražiti preko repozitorija ustanova u bazi podataka ustanova na objekt *Facility* koji ima traženi identifikator. Ako ga ne pronađe onda će vratit odgovor *ResponseEntity* sa http statusom *NOT_FOUND*, inače će pronađeni objekt *Facility* preko repozitorija ustanova izbrisati iz baze podataka ustanova te će vratiti izbrisani objekt sa http statusom *OK.*

5.8 AKCIJE REDOVA

Servis za upravljanje redovima čekanja i njegov glavni kontroler za redove sadržava javni *API* za koje nije potrebna registracija i prijava te *API* za koji je to sve potrebno. Ovaj kontroler se ponaša kao glavni iz razloga što ima akcije koje se tiču dodavanja korisnika u red, uklanjanje korisnika iz reda, povećavanje brojača reda, otvaranje reda za šalterske službenike te time i resetiranje redova i brojača i slične

pomoćne akcije za dobivanje korisnika ovisno o mailu ili dobivanje svih korisnika u nekom redu. Također servis vodi evidenciju korisnika koji čekaju u redu za određeni šalter.

Sljedeće akcije su javne gdje nije potrebno biti prijavljen na servis te se pristupaju na krajnjim točkama servisa:

- *getCurrentUserNumber: /queue/getCurrentUserNumber/{idQueue}* sa metodom zahtjeva *GET*
- *getNumberOfQueuedUsers: /queue/{idQueue}* sa metodom zahtjeva *GET*

Za akcije na redovima koje se tiču dodavanja, ažuriranja i brisanja redova u određenu ustanovu potrebno je biti registriran i prijavljen sa *ADMIN* ulogom te se pristupaju na sljedećim krajnjim točkama servisa:

- *createFacilityQueue*: */queue/create/{idFacility}* sa metodom zahtjeva *POST*
- *updateFacilityQueue*: */queue/update/{idFacility}/{idQueue}* sa metodom zahtjeva *PUT*
- *deleteFacilityQueue*: */queue/delete/{idFacility}/{idQueue}* sa metodom zahtjeva *DELETE*

Sljedeće akcije u kontroleru reda se isključivo tiču poslovne logike servisa upravljanja redova čekanja. Za pristup ovim akcijama potrebno je biti prijavljen u servis. Za određene akcije dovoljno je imati ulogu *USER* dok je za preostale potrebno imati ulogu *ADMIN*:

- *addUser: /queue/addUser/{idFacility}/{idQueue}* sa metodom zahtjeva *POST*
- *removeUser: /queue/removeUser/{idFacility}/{idQueue}* sa metodom zahtjeva *DELETE*
- *getQueuedUser: /queue/getQueuedUser* sa metodom zahtjeva *GET*
- *nextUser: /nextUser/{idFacility}/{idQueue}* sa metodom zahtjeva *POST*
- *getQueuedUserByMail: /queue/getQueuedUser/{mail}* sa metodom zahtjeva *GET*

- *getAllQueuedUsers: /queue/getAllQueuedUsers/{idQueue}* sa metodom zahtjeva *GET*
- *resetQueues: /queue/resetQueues* sa metodom zahtjeva *DELETE*

5.8.1 Dohvaćanje broja trenutnoga korisnika

Akcija dohvaćanje trenutnoga korisnika *getCurrentUserNumber* se nalazi na krajnjoj točki servisa */queue/getCurrentUserNumber/{idQueue}* te se pristupa sa *GET* metodom zahtjeva. Kontroler reda za ovu akciju očekuje obavezan identifikator *idQueue* kao parametar prilikom zahtjeva akcije budući da ta akcija vraća kao odgovor broj trenutnog korisnika u određenom redu ovisno o parametru *idQueue*. Ova akcija se odnosi na provjeravanje mape trenutnih korisnika u redu čekanja. Budući da se mapa korisnika sastoji od redova i korisnika u redu sa trenutnim brojem vrlo lako se može doći do korisnikovog broja za taj red. Upit u mapu sa proslijeđenim parametrom *idQueue*-em vrati se detalji o specifičnom redu sa uzetim brojem korisnika za taj red te servis isti taj broj vrati kao odgovor *ResponseEntity* sa http statusom *OK*. Ako upit u mapu sa proslijeđenim *idQueue*-em ne vrati rezultat, onda servis vrati odgovor NOT_FOUND.

5.8.2 Dohvaćanje broja korisnika u redu čekanja

Akcija dohvaćanje broja korisnika u redu čekanja *getNumberOfQueuedUsers* se nalazi na krajnjoj točki servisa */queue/{idQueue}* te se pristupa sa *GET* metodom zahtjeva. Servis za upravljanje redovima čekanja očekuje od zahtjeva obavezan identifikator *idQueue* kao parametar koji označava red u ustanovi. Servis prosljeđuje *idQueue* iz parametra na repozitorij korisnika u redu gdje se pretraži u bazi podataka da li postoji red sa proslijeđenim *idQueue*-jem. Ako ima onda se vraća sačuvana lista korisnika u redu te servis vraća kao odgovor *ResponseEntity* sa brojem veličine liste korisnika u redu te http statusom *OK,* inače ako je lista prazna ili ne pronađe traženi id reda u bazi podataka onda vrati odgovor sa http statusom *NOT_FOUND.*

5.8.3 Kreiranje reda ustanove

Akcija kreiranje reda ustanove *createFacilityQueue* se nalazi na krajnjoj točki servisa */queue/create/{idFacility}* te se pristupa sa *POST* metodom zahtjeva. Pristup ovoj akciji moguće je samo za prijavljene korisnike sa *ADMIN* ulogom. Kontroler reda servisa za upravljanje redovima čekanja u zahtjevu očekuje obavezan identifikator ustanove *idFacility* za koju će se red dodati. Osim toga opcionalno se dodaje naziv reda koji će se dodati u ustanovu preko *Queue* klase. Kod ove akcije redovi koji će se dodati u ustanovu bit će samo redovi sa nazivom i *id*-em. Na primjer zahtjev na krajnju točku servisa sa MUP ustanovu (npr. *id* je 1100) može izgledati ovako: */queue/create/1100?name=Putovnice*. Kontroler reda servisa prosljeđuje *idFacility* na repozitorij ustanova gdje pretražuje bazu podataka ustanova sa proslijeđenim *id*-em. Ako repozitorij pronađe ustanovu sa traženim *id*-em onda će se red iz parametra akcije dodati u ustanovu te preko repozitorija ustanova sačuvati kao takav dok će servis vratiti odgovor *ResponseEntity* sa takvim sačuvanim objektom ustanove i http statusom *CREATED*. Ako repozitorij ne pronađe ustanovu sa takvim identifikatorom onda će servis vratiti odgovor sa http statusom *NOT_FOUND*.

5.8.4 Ažuriranje reda ustanove

Akcija ažuriranje reda ustanove *updateFacilityQueue* se nalazi na krajnjoj točki servisa */queue/update/{idFacility}/{idQueue}* te se pristupa sa *PUT* metodom zahtjeva. Pristup ovoj akciji moguće je samo za prijavljene korisnike sa *ADMIN* ulogom. U ovom slučaju kontroler reda servisa očekuje obavezne parametre *idFacility* i *idQueue* koji su identifikatori za ustanovu i red tako da bi se moglo znati koja se ustanova ažurira te koji točno red u toj ustanovi. Osim toga, prosljeđuje se naziv reda na koji će se red promijeniti odnosno postavlja se atribut *name* u klasi *Queue*. Na primjer, u prethodno kreiranom redu „Putovnice“ (npr. id 101) ustanove MUP, naziv reda će se promijeniti iz „Putovnice“ u „Osobne iskaznice„: */queue/update/1100/101?name=Osobne iskaznice.* Kontroler reda servisa upravljanja redova čekanja prosljeđuje dobiveni identifikator ustanove na repozitorij ustanove gdje će repozitorij pretražiti bazu podataka ustanove sa traženim id-em te vratiti pronađenu ustanovu. Ako ne pronađe traženu ustanovu onda servis vraća odgovor *ResponseEntity* sa http statusom *NOT_FOUND*. Ako je pronašao ustanovu onda iterira po njezinim redovima te traži id reda sa proslijeđenim identifikatorom *idQueue* iz parametra metode. U slučaju da je pronašao taj red onda se mijenjaju atributi reda klase *Queue* ovisno o proslijeđenim

atributima na pozivu akcije gdje bi iz prethodnog primjera to bio atribut *name*. Nakon uspješnog ažuriranja preko repozitorija ustanova se ažurira kao takva te servis vrati odgovor sa ažuriranom ustanovom i http statusom *OK*.

5.8.5 Brisanje reda ustanove

Akcija brisanja reda ustanove *deleteFacilityQueue* se nalazi na krajnjoj točki servisa */queue/delete/{idFacility}/{idQueue}* te se pristupa sa *DELETE* metodom zahtjeva. Pristup ovoj akciji moguće je samo za prijavljene korisnike s *ADMIN* ulogom. Također i ovdje servis za upravljanje redovima čekanja očekuje obavezne parametre *idFacility* i *idQueue* koji su identifikatori ustanova i redova tako da bi se moglo znati koji će se red u kojoj ustanovi obrisati. Kontroler reda prosljeđuje identifikator ustanove na repozitorij ustanove te ako takva ustanova postoji u bazi podataka ustanove onda se vrati tražena ustanova. Ako se ustanova ne pronađe, servis vrati *ResponseEntity* sa http statusom *NOT_FOUND.* Na pronađenoj ustanovi servis će iterirati po listi redova gdje će pokušati naći red sa identifikatorom *idQueue* koji je proslijeđen kao parametar na akciji. Ako ga pronađe onda će ga izbrisati, inače će servis vratiti odgovor s http statusom *NOT_FOUND.* Nakon što je red izbrisan, servis će preko repozitorija ustanove ažurirati ustanovu u bazi podataka ustanove te će vratiti odgovor *ResponseEntity* sa ažuriranom ustanovom i http statusom *OK.*

5.8.6 Dodavanje korisnika u red čekanja

Akcija brisanja reda ustanove *addUser* se nalazi na krajnjoj točki servisa */queue/addUser/{idFacility}/{idQueue}* te se pristupa sa *POST* metodom zahtjeva. Ova akcija je glavni dio servisa koja omogućuje osnovnu funkcionalnost trenutnom prijavljenom korisniku dodavanja u izabrani red čekanja. Da bi se korisnik mogao dodati u određeni red čekanja, servis za upravljanje redova čekanja očekuje od poziva akcije da se proslijede parametri *idFacility* i *idQueue* koji su identifikatori ustanove i reda u bazi podataka. Kontroler reda servisa uzima iz sigurnosnog konteksta aplikacije trenutno prijavljenog korisnika te ga proslijedi repozitoriju korisnika da bi se dobile sve potrebne informacije o korisniku. Osim toga, servis će proslijediti identifikatore ustanove i reda iz parametra metode na repozitorije ustanova i reda te će se pretražiti

baza podataka ustanova i reda da bi se dobili objekti točno traženih ustanova i reda. Također servis će pretražiti sa dobivenim korisničkim informacijama da li slučajno već postoji taj korisnik u redu čekanja gdje će preko repozitorija za redove čekanja zatražiti listu redova u kojima korisnik čeka. Ako se slučajno ne mogu dobiti informacije o korisniku u obliku objekta *User* ili servis ne može pronaći traženu ustanovu *Facility* onda servis vraća odgovor *ResponseEntity* sa http statusom *NOT_FOUND*.

Ako servis nakon provjere da li već postoji red čekanja u kojemu korisnici čekaju ustanovi da nema nikoga u tom redu onda se kreira traženi objekt *QueuePriority* te se uveća prioritet za jedan. Taj objekt *QueuePriority* se pridruži kreiranom i nadopunjenom objektu *QueuedFacility*. U slučaju da je već postojao objekt *QueuePriority* jer već ima korisnika koji čekaju u redu onda se samo poveća prioritet za jedan.

Nakon toga servis uzima pronađenog ili novog korisnika koji čeka u redu čekanja preko objekta *QueuedUser* te provjeri da li već čeka u ustanovi proslijeđenom iz parametra akcije gdje ga nadoda na red čekanja za tu ustanovu ako ga nema, a ako već čeka u toj ustanovi onda servis provjeri u kojem redu čeka. Ako čeka već na nekom drugom redu u toj ustanovi odnosno šalteru onda se nadoda na novo traženi red čekanja u toj ustanovi. Na primjer ako netko čeka u MUP-u na šalteru za „Putovnice“ dozvoljeno je istoj toj osobi da istovremeno čeka i na šalteru za „Osobne iskaznice“. Trenutno je prepušteno korisniku na procjenu da li mu se isplati paralelno čekati na dva ili više reda te da li će stići na sve. Ako se dogodi da se korisnik pokušao dodati u red čekanja u kojem već čeka onda servis vraća odgovor *ResponseEntity* sa http statusom *CONFLICT*.

Ukoliko se korisnik uspješno dodao u red čekanja onda se perzistiraju svi podaci što se tiču korisnika i čekanja u redu pomoću odgovarajućih repozitorija u bazu podataka a to su objekti *UserInQueue*, *QueuePriority*, *QueuedUser*. Kao rezultat servis za upravljanje redovima čekanja vraća odgovor *ResponseEntity* sa http statusom *OK* te sa postavljenim objektom *QueuedUser* koji daje informaciju u kojim se sve ustanovama i redovima čekanja određeni korisnik nalazi.

5.8.7 Uklanjanje korisnika iz reda čekanja

Akcija brisanja reda ustanove *removeUser* se nalazi na krajnjoj točki servisa */queue/removeUser/{idFacility}/{idQueue}* te se pristupa sa *DELETE* metodom zahtjeva. Slično kao i kod dodavanje korisnika u red čekanja ova akcija očekuje identifikatore ustanove *idFacility* i reda *idQueue* da bi servis mogao znati iz koje se točno ustanove i reda korisnik pokušava ukloniti. Bitno je za napomenuti da neovisno o tome što se korisnik uklanja iz reda nikada se neće smanjivati prioritet u redu iz razloga da se ne poremeti prioritet sa onima koji čekaju u redu iza njega. Ova akcija omogućuje uklanjanje iz reda čekanja samo za prijavljene korisnike u sustavu. Kontroler reda servisa za upravljanje redovima čekanja uzima iz sigurnosnog konteksta aplikacije trenutno prijavljenog korisnika te provjeri da li taj isti korisnik čeka u nekom redu pomoću repozitorija korisnika u redovima čekanja. Ako nema trenutno prijavljenog korisnika u redu čekanja onda servis vrati odgovor *ResponseEntity* sa http statusom *NOT_FOUND.* Ako trenutni korisnik čeka u redu čekanja onda će servis pretražiti u kojem točno redu čeka te ga izbrisati pomoću repozitorija korisnika u redovima čekanja iz baze podataka. Osim toga ukloniti će se iz pomoćnog repozitorija koji upravlja sa objektom *QueuedUser* te je tom akcijom osigurano da je korisnik sigurno uklonjen iz željenoga reda čekanja. Servis će vratiti odgovor *ResponseEntity* sa http statusom *OK* te sa izmijenjenim objektom *QueuedUser* gdje će se vidjeti brisani red kod korisnika u redu čekanja.

5.8.8 Dohvaćanje korisnika u redu čekanja

Akcija dohvaćanje korisnika u redu čekanja *getQueuedUser* se nalazi na krajnjoj točki servisa */queue/getQueuedUser* te se pristupa sa *GET* metodom zahtjeva. Ova akcija uzima trenutnog prijavljenog korisnika iz sigurnosnog konteksta aplikacije te pretražuje bazu podataka pomoću repozitorija korisnika u redu čekanja. Ako servis za upravljanje redovima čekanja ne pronađe korisnika u redu čekanja onda se vrati odgovor *ResponseEntity* sa http statusom *NOT_FOUND*, inače vrati odgovor sa http statusom *OK* i pronađenim objektom *QueuedUser* koji prikazuje u kojima sve redovima i ustanovama trenutni korisnik čeka.

5.8.9 Sljedeći korisnik

Akcija sljedeći korisnik *nextUser* se nalazi na krajnjoj točki servisa */queue/nextUser/{idFacility}/{idQueue}* te se pristupa sa *POST* metodom zahtjeva. Ova akcija je dostupna samo korisnicima sa *ADMIN* ulogom. Servis za upravljanje redovima čekanja očekuje od akcije parametre identifikatore *idFacility* za ustanove i *idQueue* za redove da bi mogao znati u kojoj ustanovi i u kojem redu treba uzeti sljedećeg korisnika na šalter. Kontroler reda servisa će pomoću identifikatora *idQueue* preko repozitorija korisnika u redu čekanja zatražiti listu takvih korisnika iz baze podataka. Ako je lista prazna onda će servis vratiti odgovor *ResponseEntity* sa http statusom *NOT_FOUND* što znači da nema korisnika koji čekaju na tom specifičnom redu. Ako lista nije prazna onda servis uzima sa čela liste korisnika koji čeka u redu čekanja, odnosno korisnika koji je prvi na redu te ga se dodaju u mapu trenutnih korisnika u obradi. Nakon što je servis prebacio korisnika u redu čekanja u trenutnu obradu istovremeno je potrebno da se izbriše iz baze podataka korisnika u redu čekanja gdje servis to odradi uz pomoć dodijeljenoga repozitorija. Može se spomenuti da je postupak uklanjanja korisnika iz reda čekanja u ovoj akciji prilikom uzimanja sljedećeg korisnika jednak dijelu akcije uklanjanju korisnika iz reda čekanja. Nakon uspješnog prebacivanja prvog korisnika sa čela lista u mapu trenutnih korisnika u obradi te brisanja tog korisnika iz redova čekanja, servis vrati odgovor *ResponseEntity* sa http statusom *OK.*

5.8.10 Dohvaćanje korisnika u redu čekanja pomoću maila

Akcija dohvaćanje korisnika u redu čekanja pomoću maila *getQueuedUserByMail* se nalazi na krajnjoj točki servisa */queue/getQueuedUser/{mail}* te se pristupa sa *GET* metodom zahtjeva. Ova akcija je namijenjena za administracijski alat te je shodno tome potrebna *ADMIN* uloga prijavljenog korisnika. Servis očekuje kao parametar akcije *mail* korisnika da bi se ona mogla izvršiti. Kontroler reda servisa zatim prosljeđuje *mail* iz parametra akcije na repozitorij korisnika u redu čekanja te ako ga nije pronašao servis vraća odgovor *ResponseEntity* sa http statusom *NOT_FOUND*, inače vrati pronađenog korisnika u redu čekanja gdje ga servis proslijedi kao odgovor i http status *OK*.

5.8.11 Dohvaćanje svih korisnika u redu čekanja

Akcija dohvaćanje svih korisnika u redu čekanja *getAllQueuedUser* se nalazi na krajnjoj točki servisa */queue/getAllQueuedUser/{idQueue}* te se pristupa sa *GET* metodom zahtjeva. Izvršavanje ove akcije moguće je samo korisnicima koji imaju *ADMIN* ulogu. Servis očekuje obavezan parametar *idQueue* koji je identifikator reda u bazi podataka. Kontroler reda servisa prosljeđuje identifikator iz parametra akcije na repozitorij korisnika u redu čekanja gdje se vrati lista korisnika u redu čekanja za taj proslijeđeni red. Ako je lista prazna onda servis vrati odgovor *ResponseEntity* sa http statusom *NOT_FOUND*, inače vrati kao odgovor dobivenu listu korisnika u redu čekanja te http status *OK.*

5.8.12 Resetiranje brojača svih redova

Akcija resetiranje brojača svih redova *resetQueues* se nalazi na krajnjoj točki servisa */queue/resetQueues* te se pristupa sa *DELETE* metodom zahtjeva. Pristup ovoj akciji imaju samo korisnici sa *ADMIN* ulogom. Kontroler reda servisa za upravljanje redovima čekanja pomoću repozitorija redova iterira sve redove te resetira njihove prioritete na nulu te takve sačuva u bazi podataka redova. Nakon uspješnog izvršavanja resetiranja servis vrati odgovor *ResponseEntity* sa listom resetiranih redova i http statusom *OK.*

ZAKLJUČAK

Čekanje u redu je svima frustrirajuće, pogotovo kada se mora čekati jer je obavezno da se posao obavi sada, a potrebno je obići još ustanova ili se znatno prekoračilo planirano vrijeme zbog čekanja u redu. Nakon mnogo primjera čekanja u redu prirodno je zapitati se za neko rješenje o pokušaju umanjivanja ili rješavanja takvog problema kojom se dolazi do ideje o korištenju IT tehnologije. Tehnologija današnjice se sve više okreće mobilnom svijetu što dodatno daje ideje o načinu implementacije sustava. Realno je za shvatiti da se ne može praktično napraviti sustav koji bi u potpunosti riješio problem redova čekanja, ali je moguće napraviti sustav koji će njime upravljati te time i smanjiti vrijeme provedeno čekajući u redu. Osim toga bitno je da je sustav neovisan o platformi, vrsti uređaja i uvijek dostupan te je zbog toga cilj bio izraditi web servis sa *API*-em kojem se može pristupiti mobilnom ili web aplikacijom ili nekim drugim servisom.

Za ispunjenje cilja ovog diplomskog rada bilo je nužno istražiti potrebne tehnologije koje bi najbolje odgovarale za elegantnu i brzu izradu web servisa koje se mogu primijeniti na stvarnom tržištu. Iz toga razloga se koristio Java programski jezik i „Spring Boot“ programsko radno okruženje za brzu i efikasnu izradu REST servisa. Među ostalim, MongoDB baza podataka se savršeno uklopila u servis koji je u ranoj fazi izrade, sa tendencijom česte izmjene ili naglog skaliranja, dok korištenje autentikacije preko OAuth2 standarda se postiže visoka razina sigurnosti te omogućuje da servis sve više nalikuje jednom kvalitetnom proizvodu koji može konkurirati na tržištu.

Svaki IT proizvod se sastoji od procesa dugotrajnog razvijanja novih funkcionalnosti, poboljšavanja starih i popravljanja nastalih pogrešaka. Tako da ni izrađeni web servis za upravljanje redovima čekanja nije savršen proizvod, te su moguća brojna poboljšanja. Smatram da je jedno od korisnijih poboljšanja predviđanje vremena potrebnog do dolaska na red gdje bi se sa time olakšalo korisnicima da preciznije planiraju vrijeme te komotnije dočekaju svoj red. Osim toga, moglo bi se napraviti sustav za automatsku regulaciju otvorenih šaltera tako da pozove po potrebi više šalterskih službenika ili manje kada to nije potrebno.

LITERATURA

Hardt, D., 2012. *The OAuth 2.0 Authorization Framework.* [Mrežno]
Available at: https://tools.ietf.org/html/rfc6749
[Pokušaj pristupa 21 Svibanj 2018].

Heroku, 2018. *About Heroku.* [Mrežno]
Available at: https://www.heroku.com/about
[Pokušaj pristupa 13 Svibanj 2018].

mLab, 2018. *About mLab.* [Mrežno]
Available at: https://mlab.com/company/
[Pokušaj pristupa 13 Svibanj 2018].

Oracle, 2013. *The Java EE 6 Tutorial.* [Mrežno]
Available at: https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html
[Pokušaj pristupa 30 Travanj 2018].

Quora, 2018. *Why use Intellij IDEA over Eclipse?.* [Mrežno]
Available at: https://www.quora.com/Why-use-Intellij-IDEA-over-Eclipse
[Pokušaj pristupa 10 Svibanj 2018].

Sarig, M., 2017. *CouchDB vs MongoDB.* [Mrežno]
Available at: https://blog.panoply.io/couchdb-vs-mongodb
[Pokušaj pristupa 29 Travanj 2018].

Singh, R. K., 2017. *Reduce Java's boilerplate code using Project Lombok.* [Mrežno]
Available at: https://www.callicoder.com/reduce-java-boilerplate-code-using-project-lombok/
[Pokušaj pristupa 13 Svibanj 2018].

solid IT, 2018. *DB-Engines Ranking of Document Stores.* [Mrežno]
Available at: https://db-engines.com/en/ranking/document+store
[Pokušaj pristupa 29 Travanj 2018].

Trelle, T., 2012. *Spring Data – One API To Rule Them All?.* [Mrežno]
Available at: https://www.infoq.com/articles/spring-data-intro
[Pokušaj pristupa 15 Svibanj 2018].

Tutorials Point (I) Pvt. Ltd., 2018. *OAuth 2.0 - Refresh Token.* [Mrežno]
Available at: https://www.tutorialspoint.com/oauth2.0/refresh_token.htm
[Pokušaj pristupa 27 Svibanj 2018].

Tutorials Point (I) Pvt. Ltd., 2018. *What are Web Services?.* [Mrežno]
Available at:
https://www.tutorialspoint.com/webservices/what_are_web_services.htm
[Pokušaj pristupa 10 Svibanj 2018].
Walls, C., 2016. *Spring Boot in Action.* New York: Manning Publications Co..
Webb, P., 2013. *Spring Boot – Simplifying Spring for Everyone.* [Mrežno]
Available at: https://spring.io/blog/2013/08/06/spring-boot-simplifying-spring-for-everyone/
[Pokušaj pristupa 20 Travanj 2018].
Yegulalp, S., 2017. *What is NoSQL? NoSQL databases explained.* [Mrežno]
Available at: https://www.infoworld.com/article/3240644/nosql/what-is-nosql-nosql-databases-explained.html
[Pokušaj pristupa 28 Travanj 2018].

POPIS SLIKA

POPIS TABLICA

SAŽETAK

U ovom radu se istražuju tehnologije koje bi najbolje odgovarale za brzu i jednostavnu izradu REST servisa za upravljanje redovima čekanja na produkcijskoj razini u Java programskom jeziku. Na temelju istraživanja, odabrane tehnologije su se primijenile u samoj izradi REST servisa te je objašnjeno što je nužno imati za funkcionalni servis.

Objašnjeno je korištenje i primjena „Spring Boot“ programskog okvira pomoću generatora predložaka „Spring Initializr“, „MongoDB“ baze podataka sa povezivanjem entiteta na servisu pomoću „Spring Data“ te konfiguriranja i implementiranja „Spring Security“ OAuth2 standarda na krajnje točke servisa.

REST servis za upravljanje redovima čekanja je napravljen s osnovnim funkcionalnostima, a da bude primjenjiv na nekom realnom slučaju, odnosno ustanovi. Na kraju, dana su poboljšanja servisa koja mogu unaprijediti planiranje vremena korisnika i preciznije odrediti otvaranje šaltera.

Ključne riječi: REST, web servis, Java, Spring Boot, redovi čekanja

SUMMARY

This thesis researches technologies that would be the best for fast and easy development of production level queue managing REST service in the Java programming language. Based on the research, chosen technologies were used in development of REST service and explained what is necessary to have for functional service.

Also, it was explained the utilization and implementation of the "Spring Boot" framework by using template generator "Spring Initializr", "MongoDB" database by linking service entities with the help of „Spring Data“, configuring and implementing the "Spring Security" OAuth2 standard on service's endpoints.

Queue managing REST service is made with basic functionality to be applicable in a real case scenario, for example an institution. Finally, some service improvements were given that could help improving customer time planning and more precisely determine the opening of a counter.

Keywords: REST, web service, Java, Spring Boot, waiting queues